Modular subclass verification:
Safely creating correct subclasses without superclass code

Clyde Dwain Ruby
TR #06-34
December 2006

Keywords: Downcalls, super-calls, subclass, semantic fragile subclassing problem, subclassing contract, code contract, specification inheritance, method refinement, alias control, specification of side effects, Java language, JML language.
2006 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — Languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — Object-oriented design methods, software libraries; D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract, reliability, validation, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation, extensibility; D.2.10 [*Software Engineering*] Design — Methodologies, tools, JML; D.2.11 [*Software Engineering*] Software Architectures — Data abstraction, information hiding, languages, JML; D.2.13 [*Software Engineering*] Reusable Software — Reusable libraries; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented langauges; D.3.3 [*Programming Languages*] Language Constructs and Features — Classes and objects, frameworks, inheritance; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques.

Copyright © Clyde Dwain Ruby, 2006.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Modular subclass verification:

Safely creating correct subclasses without superclass code

by

Clyde Dwain Ruby

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee
Gary T. Leavens, Major Professor
Samik Basu
Clifford Bergman
Shashi K. Gadia
Jonathan D. H. Smith

Iowa State University

Ames, Iowa

2006

Copyright © Clyde Dwain Ruby, 2006. All rights reserved.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my advisor Gary T. Leavens for his support and guidance throughout this journey which is now finally coming to a conclusion. He spent many hours teaching me how to improve my technical writing skills; specifically, his feedback on my writing of this dissertation was invaluable. I will always be grateful that Gary always seemed to find time to discuss my ideas and give direction whenever I needed that. I always looked forward to our twice weekly phone meetings during these last few years when I was living away from Ames; he even allowed me to call him at home and on weekends when I had something urgent to discuss. I will also fondly remember the times Gary and his wife Janet opened their home to my wife and I when we needed to be back in Ames. Gary is not only an excellent mentor but a valued friend.

This research was only possible because of the work of others before me. In particular, the work of K. Rustan M. Leino paved the way for my research. My research also builds on the work of others, most notably Arnd Poetzsch-Heffter, Peter Mueller, and Gary Leavens.

I also wish to thank the other students in the JML research group at Iowa State University; we had many interesting and productive discussions. I would particularly like to thank Curtis Clifton and Yoonsik Cheon who provided valuable assistance when I was first learning the details of our second implementation of the JML tool; this made it easier for me to add the features related to my work. Thanks also to David Cok, an independent contributor to the JML project, for helpful comments on an earlier version of Chapter 2.

I am also indebted to the excellent faculty of Iowa State University, and particularly, Jack Lutz, Clifford Bergman, Jonathan Smith, and Gary Leavens whose courses provided practical skills in logical and critical thinking that were so necessary for the construction of a large proof. Special thanks also to the administration and Computer Science Department of Maharishi University of Management for their encouragement and support.

I would also like to thank Rockwell Collins and the Graduate College who provided scholarships and financial support during my time at Iowa State. This research was also supported in part by grants from the National Science Foundation.

Finally, I am especially grateful to my wife Susan who somehow had the patience to continually encourage and support me all the way through. Without her understanding and empathy during this time, this research would not have come to fruition. Thanks also to our families who provided me with very memorable and necessary breaks from the focus of this research.

ABSTRACT

The documentation of object-oriented frameworks and class libraries needs to provide enough information so programmers can reason about the correctness of subclass methods without superclass code. Even though a superclass method satisfies its specification and behaves properly in the context of the superclass itself, a new subclass may cause that method to have unexpected or unverifiable behavior. For example, inherited superclass code can call down to subclass methods which may cause a superclass method to no longer satisfy its specification. Furthermore, without superclass code, downcalls can result in unverifiable side-effects. Aliasing can also result in unexpected or unverifiable side-effects.

In this dissertation, we present a reasoning technique that allows programmers, who have no access to superclass code, to avoid the problems caused by downcalls and aliasing. The rules use the specification of the abstract data representation of a class and the frame axiom of each method to determine when a method override is necessary and when verifiers can safely reason about the behavior of super-calls.

We describe a type system and propose a tool that would warn when a super-call is unsafe or when a superclass method needs to be overridden. A verification logic is also presented and proved sound. The verification logic is based on specifications given in the Java Modeling Language (JML) and uses superclass and subclass specifications to modularly verify the correctness of subclass code.

A set of guidelines is proposed for class library implementers that, if followed, guarantees that superclass methods will always be safe to call and that our verification logic can safely be used. These guidelines make our technique easy to use in practice.

CHAPTER 1: INTRODUCTION

1.1 Background

One major benefit of object-oriented (OO) frameworks and class libraries is software reuse. For example, one or more components of a framework or class library may provide a functionality similar to the requirements of a new system. In such situations, rather than completely rewriting the code, classes in the library can be reused, customized, and recombined to provide the new functionality and without changing library source code. Frameworks and class libraries have two kinds of reusers, *clients* who simply use the classes in the framework or library in their programs, and *customizers* who extend and override parts of these classes to customize them for a specific application. An important requirement for the customizer of a framework is the ability to write new code and then combine it with the existing code in such a way that a new implementation is created. The mechanism that allows such combining is inheritance, where the subclass inherits method and field definitions from the superclass. In addition, the customizer can adapt and extend the behavior of the superclass by overriding superclass methods and adding new methods to the new subclass.

Documentation plays an important role in the reuse of frameworks and class libraries. For clients, software reuse depends on understanding the behavior of public methods and constructors of existing classes in the framework. In contrast, customizers not only need an understanding of the behavior of public components but also the behavior of protected components and how public and protected components interact. For example, overriding one method of a class may also require overriding other methods that interact with this method [KL92, Lam93, RL00].

Furthermore, documentation of the superclass needs to be precise enough so that clients and customizers can reuse and modify the framework. At the same time, it should not be overly specific, thereby limiting the possible implementations of superclasses. For example, documentation should allow library implementations to be improved or made more efficient without affecting the correctness of subclass implementations. Moreover, the documentation should allow modular reasoning about the behavior of superclasses and subclasses, that is, without knowing the contexts where these classes will be used and without requiring that superclasses know anything about their subclasses.

1.2 The Problem

A long term goal of the research presented in this dissertation is to discover what makes good quality documentation of OO frameworks and class libraries. In addition to the above mentioned characteristics, the goal has been to discover what information is needed by programmers so library classes can safely be extended even when the framework or library provider does not make the superclass code available. Usually programmers must study the library's source code when writing subclasses, especially if unexpected problems arise, because frameworks and class libraries do not provide enough documentation. One benefit of sufficient and unambiguous documentation would be to

allow companies to protect their investment in source code. In particular, this would allow a company to only ship compiled code and documentation.

The formal notations and reasoning systems developed and described in this dissertation provide an understanding of some of the requirements of quality documentation. The reasoning technique is formalized with a set of rules and some new forms of specification. The technique demonstrates what information is necessary in library documentation, and how to use it to create subclasses that do not exhibit problems such as non-termination or unexpected behavior. The specification and reasoning technique allows programmers, who have no access to the code of the superclass, to determine when it is safe for a subclass to make a superclass call, and when a method must be overridden to avoid potential problems [RL00].

Because the rules assume the superclass code is unavailable, they also provide another benefit; they specify the conditions under which modular verification of subclass methods is possible. A subclass method can be *verified modularly* if it can be verified using only the implementation code of the subclass and the specifications of the classes in scope. If the rules in our system are violated, as will be illustrated by the examples that follow (here and in Chapters 2 and 3), then in general the superclass code is needed for sound verification.

Furthermore, Perry and Kaiser [PK90] point out that inherited superclass methods must be retested unless "the new subclass is a pure extension of the superclass, that is, … there are no interactions in either direction between the new [subclass] instance variables and methods and any inherited instance variables and methods." They further show that a different set of tests may be needed to retest these inherited methods. However, our technique identifies the interactions between superclasses and subclasses that create problems. So if the documentation and reasoning technique we propose is followed, then inherited methods would not need to be retested in the context of the new subclass.

The reasoning system uses some new forms of specification that are being embodied in the Java Modeling Language (JML) [LBR98, LBR01], a specification language tailored to Java programs. Although the ideas and techniques are intended to be independent of any particular programming language, for concreteness a subset of Java and JML will be used as the programming and specification notations.

A further goal of the research has been to develop a tool to automatically extract the information needed to apply the formal rules. This tool would also be used to aid in reasoning about the correctness of subclasses by checking the specifications of subclasses and enforcing the formal rules.

Another goal has been to determine how framework and class library designers and implementers can simplify the reasoning problems of reusers. To this end, the formal rules have been used as the basis of a set of guidelines for library designers and implementers; [RL00] shows that such guidelines can significantly simplify the reasoning of reusers, and in particular customizers of a class library.

The research so far has shown that framework and class library documentation needs to include at least four things.

1. It has to include the usual documentation specifying the functional behavior of public methods in the library so library classes and their subclasses can safely be used in client code.
2. It has to include documentation of the behavior of protected methods and data fields so customizers can reuse methods and data visible in subclasses but without seeing the superclass code.
3. It needs to include the method calling structure and variables accessed (read) by methods of library classes so programmers can safely override superclass methods without studying the superclass code (see Section 1.4) [RL00].
4. It needs to specify when a method modifies protected and private fields (i.e., fields not visible to clients of a class) so customizers can prevent unexpected side-effects to internal objects that disallow sound, modular reasoning (see Section 1.5).

The examples in the following sections illustrate why this kind of information is needed in documentation.

1.3 Specification of Method Behavior

Programmers need to know the behavior of methods in a framework or class library so they can use the library classes in client programs. Furthermore, behavioral specifications are needed so programmers can create behavioral subtypes when inheriting from and extending these classes. *Behavioral subtyping* is based on the behavior of types and ensures that subtype objects behave like supertype objects. Hence, behavioral subtype objects can be used in place of supertype objects without producing any unexpected behavior [Ame91, DL96, LW93, LW94]. A subclass implements a behavioral subtype if each overriding subclass method refines the method it overrides. Method refinement is a relationship between behavioral specifications. The basic idea is that specification *B* *refines* specification *A* if the allowed behavior of *B* is a subset of the allowed behavior of *A* [Ame91, BvW98, Heh93, Mor94]. A practical definition of method refinement is given in Figure 1.1[1] (a more extensive discussion is found in Section 2.9.1).

1.3.1 Public Specifications

Figure 1.2 shows a formal public JML specification of a simple `IntCell` class. As a public specification, it documents the behavior of public methods and instance variables, and therefore protected and private members are not included. This is the kind of information that would be provided to clients of the class.

---

1. This definition is equivalent when proof obligation 2 of Figure 1.1 is replaced with the following [CC00, DL96]:

   2'. $Theory(S) \vdash (\backslash\mathbf{old}(Pre^m{}_S) \Rightarrow Post^m{}_S) \Rightarrow (\backslash\mathbf{old}(Pre^m{}_C) \Rightarrow Post^m{}_C)$

Suppose *S* is a subtype of *C*, then the public and protected specification of method *m* in *S* refines that of method *m* in *C* if

1. *Theory(S)* |- **\old**($Pre^m{}_C$) ⇒ **\old**($Pre^m{}_S$)
2. *Theory(S)* |- **\old**($Pre^m{}_C$) ⇒ ($Post^m{}_S$ ⇒ $Post^m{}_C$)
3. *Theory(S)* |- $Assignable^m{}_S$ ⊆ $Assignable^m{}_C$

Figure 1.1: A practical notion of method refinement [Ame91, BvW98, Mey88, Mor94].

JML blends the Eiffel [Mey97] and Larch [GHG+93] traditions, and following Eiffel, uses Java expressions within assertions. JML behavioral specifications are found between the annotation markers `/*@` and `@*/` and on lines starting with `//@`. The JML tools distinguish between ordinary comments and JML specifications via the "`@`" that follows the comment delimiters "`//`" and "`/*`" at the beginning of a JML specification (see Figure 1.2). Because JML specifications are included inside Java comments, they are ignored by the Java compiler; this allows specifications to be embedded in Java programs. Within multi-line annotations, initial at-signs (`@`) on a line are ignored. Method specifications appear before the method header declaration.

JML also permits methods to be called in specifications, such as in the pre- or postcondition of a requires or ensures clause. However, only methods without side-effects may be called in these assertions; this is specified using the pure modifier. That is, the modifier pure specifies that a method has no side-effects, and thus could be used in assertions.

The modifier model in a declaration means that the declaration is for use in specifications and need not be part of the implementation[2]. For example, in Figure 1.2, `value` is an instance variable used only for specifying method behavior. Note also that the declaration of `value` is enclosed entirely in a JML annotation. In JML method specifications, preconditions are preceded by the keyword requires, postconditions with ensures, and frame conditions with assignable. An assignable clause specifies the variables that may be modified by the method[3]; for example, in the method `setValue`, only model variable `value`, and, implicitly, the concrete variables in its associated data group (see subsection 1.4.1 below) may be changed.[4]

2. Model variables are similar to ghost or auxiliary variables.
3. The assignable clause is similar to the frame axiom in other specification languages. However, the assignable clause is stronger than the frame axiom in that it requires that variables be listed even when they are temporarily assigned and then restored to their original value before exiting the method (see assumptions in subsection 1.6.6).
4. In JML, the modifiable keyword is a synonym for assignable, and also specifies the same prohibition against assigning to variables not listed in the clause.

```
public class IntCell {
   //@ public model int value;      // model variable

  /*@  public normal_behavior
    @     assignable value;
    @     ensures value == initVal;   @*/
  public IntCell(int initVal);

  /*@  public normal_behavior
    @     ensures \result == value;   @*/
  public /*@ pure @*/ int getValue();

  /*@  public normal_behavior
    @     assignable value;
    @     ensures value == newVal;   @*/
   public void setValue(int newVal);

  /*@  public normal_behavior
    @     requires c != null;
    @     assignable value;
    @     ensures value == c.value;  @*/
   public void setFrom(IntCell c);

  /*@   also
    @   public normal_behavior
    @     requires c instanceof IntCell;
    @     ensures !\result || (this.value == ((IntCell)c).value);
    @   also
    @   public normal_behavior
    @     requires !(c instanceof IntCell);
    @     ensures \result == false;   @*/
   public /*@ pure @*/ boolean equals(Object c);
}
```

Figure 1.2: `IntCell`'s public specification from file `IntCell.jml-refined`. JML behavioral specifications are found on lines starting with `//@` and between the annotation markers `/*@` and `@*/`. Within annotations, an initial at-sign (`@`) on a line is ignored.

The pseudo-variable \result specifies the return value of a method as in the postconditions of methods `getValue` and `equals`.

The "normal" in the normal_behavior keyword indicates that the constructor or method must not signal exceptions, and thus, when its precondition is satisfied, its final state must satisfy the corresponding postcondition. Also, the public modifier means that the specification is a public

specification, and thus only public variables and methods are in scope; for example, in a public specification, no protected variables can be mentioned.

To make specifications easier to read, JML allows them to be divided into several specification cases. *Specification cases* are specifications separated by the keyword also. For example, the `equals` method of Figure 1.2 has two specification cases. Furthermore, when a method specification has more than one case, then all must be satisfied by that method's implementation. Moreover, preconditions can overlap. When the preconditions of more than one specification case hold at the same time, then the postconditions of these overlapping cases must all be satisfied.

Furthermore, when the entire method specification is preceded by the also keyword, as in the specification of `equals` in Figure 1.2, it means that we are extending that method's specification with additional cases. In this example, these two specification cases have to be satisfied in addition to any inherited from a supertype, i.e., from `Object`. These two cases do not overlap, but they will overlap with the specification cases inherited from `Object`.

1.4 Specification of the Method Calling Structure

The following subsections give a simple example to illustrate how downcalls can cause problems when trying to reason about the behavior of subclass methods. The example also shows why the calling structure of methods is needed in specifications.

A *downcall* occurs when a superclass method directly or indirectly calls a method that has been overridden by the subclass. We also say that the superclass "calls down to" the overriding method, because we visualize the class diagram with subclasses below superclasses. Figure 1.3 shows an example of a downcall introduced by class `B` when it overrides method `m2`. When method `m1` is called on instances of class `B`, `m1` calls `m2` of the subclass rather than `m2` of class `A`. Downcalls and method overrides are important because, in a correct implementation, problems such as non-termination or unexpected behavior do not occur when a superclass calls one of its own methods. On the other hand, when the superclass calls down to an overriding method, the called method may behave differently than the superclass method expects.

Downcalls are also related to callbacks [SGM02](p. 120). A *callback* occurs when a method in class `A` calls a method of some other class `B` which then directly or indirectly calls back to a method of class `A`. In the presence of downcalls, callbacks happen, for example, when a subclass method calls a superclass method that then makes a downcall leading back to the subclass. Similarly, a superclass method can make a downcall to a subclass method that calls back to a superclass method. However, not every downcall results in a callback. For example, Figure 1.3 shows a superclass calling down to an overriding method, but the subclass does not call any superclass methods.

```
public class A {
    public int m1() { ...  m2();  ... }
    public void m2() { ... }
}

public class B extends A {
    public void m2() { ... }
}
```

Figure 1.3: An example of a downcall introduced by the override of method `m2` in subclass `B`.

1.4.1 Subclasses and Specification Inheritance

Figure 1.4 shows the public specification of class `CellPlusTotal`; it is a subclass of `IntCell` and adds the instance variable `totalChg` to keep track of the total changes made to the cell's value since its instantiation. This subclass also adds method `getTotalChange` and adds to the specifications for `setFrom` and `setValue`. In JML, a subclass inherits method specifications from its superclasses and the interfaces it implements. Therefore, any overriding subclass method, like `setFrom` or `setValue`, must satisfy the specification of the superclass method it overrides. In JML, when a subclass method overrides a method of a superclass, its additional specification is introduced by the keyword also. The expression \old(`e`) means evaluate expression `e` in the pre-state as, for example, in the postcondition of `setValue` and `setFrom`.

To satisfy the specification of `CellPlusTotal` the methods `setFrom` and `setValue` must be able to assign to `totalChg`. However, the method specifications inherited from class `IntCell` appear to only allow `value` to be assigned. Such situations are resolved in JML using the concept of data groups [Lei95, Lei98, LPHZ02].

A *data group* is an abstraction over a set of fields that are grouped together because they tend to change as a group. These groups are used to control which variables can be assigned by methods. In JML, the declaration of a model field automatically creates a data group with the same name; furthermore, the declared field is always a member of the group it creates. Typically a public model field, like `value` in Figure 1.3, is declared to represent part of an abstract value of a type. All related fields are then declared to be members of the associated data group. The members of the associated data group will include those concrete fields used to compute the abstract value of the model field.

In JML, a field or group can become a member of another group through the data group clauses that follow the field declaration. For example, in Figure 1.4 the 'in clause' says that the subclass model variable `totalChg` is a member of the data group created by the `value` model field[5]; this allows `totalChg` to be modified by a method whenever `value` is assignable. Without this clause, `totalChg`

```
public class CellPlusTotal extends IntCell {

  //@ public model int totalChg;
  //@                      in value;

  /*@  public normal_behavior
    @     assignable value, totalChg;
    @     ensures value == initVal && totalChg == 0;
    @*/
  public CellPlusTotal(int initVal);

  /*@ also
    @  public normal_behavior
    @     assignable totalChg;
    @     ensures totalChg == \old(totalChg) + Math.abs(value - \old(value));
    @*/
  public void setValue(int newVal);

  /*@ also
    @  public normal_behavior
    @     requires c != null;
    @     assignable totalChg;
    @     ensures totalChg == \old(totalChg) + Math.abs(value - \old(value));
    @*/
  public void setFrom(IntCell c);

  /*@  public normal_behavior
    @     ensures \result == totalChg;
    @*/
  public /*@ pure @*/ int getTotalChange();
}
```

Figure 1.4: `CellPlusTotal`'s public specification from the file `CellPlusTotal.jml-refined`.

cannot be assigned by any of the overriding methods of `CellPlusTotal`, and therefore without it the specification given in Figure 1.4 would not be correctly implementable. Thus, in JML, the names listed in a method's assignable clause not only denote specific fields but data groups (and their members) as well.

Notice that in this example, we need to change a field that was not in scope when the `value` model field was declared. Thus, as is often the case, not all members of a data group will be in scope when the

5. We also say in short that `totalChg` is a member of `value`.

group (i.e., model field) is first declared. However, for modularity and soundness, all data group memberships of a new field must be specified within the same context as its declaration [Lei95, Lei98, LPHZ02, LN00, LN02]; this is ensured through syntax that requires that all data group clauses immediately follow a field declaration. Thus a new field and its data group memberships can only be declared in the same context.

1.4.2 Protected Specifications

In addition, programmers implementing `CellPlusTotal` need more information about any protected instance variables and methods of the superclass. In JML, the protected specification provides the additional documentation needed. The protected specification specifies the parts of the class visible to subclasses, including protected instance variables and the behavior of protected constructors and methods. It includes information about public instance variables and methods that is not of interest to clients, but will be of interest to the implementer of a subclass. It also includes information about protected variables and methods needed to verify the correctness of the implementation of the class. Figure 1.5 gives the protected specification of class `IntCell`; it specifies the relationship between the public and protected instance variables and gives the specification of a protected constructor. The protected specifications in Figure 1.5 are combined with the public specifications in Figure 1.2 because of the refines clause found after the package declaration in Figure 1.5.

In Figure 1.5, the data group and represents clauses are used to specify a relationship between variables. These clauses are typically used to specify a relationship between variables defined in two different places as in this case where they are defined in the public and protected specifications of `IntCell`, or, as in Figures 1.2 and 1.4, in the superclass and subclass; in particular, the variables often have different visibility and different scope. The data group and represents clauses of Figure 1.5 specify the relation between the public specification-only (model) variables and the protected concrete variables of class `IntCell`. The term *concrete* means part of the implementation of a class[6]. The public variables are used for reasoning about the behavior of public methods, whereas the protected variables are needed so the implementation is not exposed to clients of the class. As described earlier, the data group clauses control which concrete variables can be assigned by method implementations.[7] In Figure 1.6, the data group clause illustrates the transitive property of the data group relation; that is, concrete variable `_totalChg` is a member of `totalChg` which is a member of `value` and therefore `_totalChg` is also a member of the `value` data group.

---

6. In this dissertation, to distinguish concrete variables from model variables, concrete variables will begin with the underbar (_) character.

7. Note that the data group clauses do not control what names are in scope in a specification.

```
//@ refines "IntCell.jml-refined";

public class IntCell {

  protected int _val;
  //@              in value;

  //@ protected represents value <- _val;

  /*@ protected normal_behavior
    @   requires c != null;
    @   assignable value;
    @   ensures this.value == c.value;
    @*/
  protected IntCell(IntCell c);
}
```

Figure 1.5: `IntCell`'s protected specification in file `IntCell.jml`.

```
//@ refines "CellPlusTotal.jml-refined";

public class CellPlusTotal extends IntCell {

   protected int _totalChg;
  //@                in totalChg;

  //@ protected represents totalChg <- _totalChg;
}
```

Figure 1.6: `CellPlusTotal`'s protected specification from `CellPlusTotal.jml`.

The represents clauses also specify a relationship between variables; they specify how a variable (typically a model field) can be derived from one or more other variables. For example, the first represents clause specifies how the value of `value` can be derived from the concrete variable `_val`, i.e., in this case they must have the same value. Note that the protected instance variable `_val` will also appear in the implementation of the class (see Figures 1.5 and 1.8) since it is not a model variable (the JML type checker emits an error message if it is not in the implementation).

The data group and represents clauses are abstraction mechanisms used to keep non-public concrete variables hidden from clients. That is, they are used so the public behavior of methods can be

```
//@ refines "CellPlusTotal.jml";

public class CellPlusTotal extends IntCell {

  protected int _totalChg;

  // ...

  public void setValue(int newVal) {
      _totalChg += Math.abs(newVal - _val);
      super.setValue(newVal)
  }
  public void setFrom(IntCell c) {
      _totalChg += Math.abs(c.getValue() - _val);
      super.setFrom(c);  // may introduce a callback
  }
}
```

Figure 1.7: A fragment of `CellPlusTotal`'s implementation from the file `CellPlusTotal.java`.

```
//@ refines "IntCell.jml";

public class IntCell {

  protected int _val;

  public void setValue(int newVal) {
      _val = newVal;
  }
  public void setFrom(IntCell c) {
      if (! this.equals(c) ) {
          setValue(c.getValue());  // possible downcall here
      }
  }
  ...
}
```

Figure 1.8: A fragment of `IntCell`'s implementation from the file `IntCell.java`.

specified without mentioning concrete variables [Lei95]. Therefore, programmers can implement the specified public behavior while keeping the underlying data structures hidden.

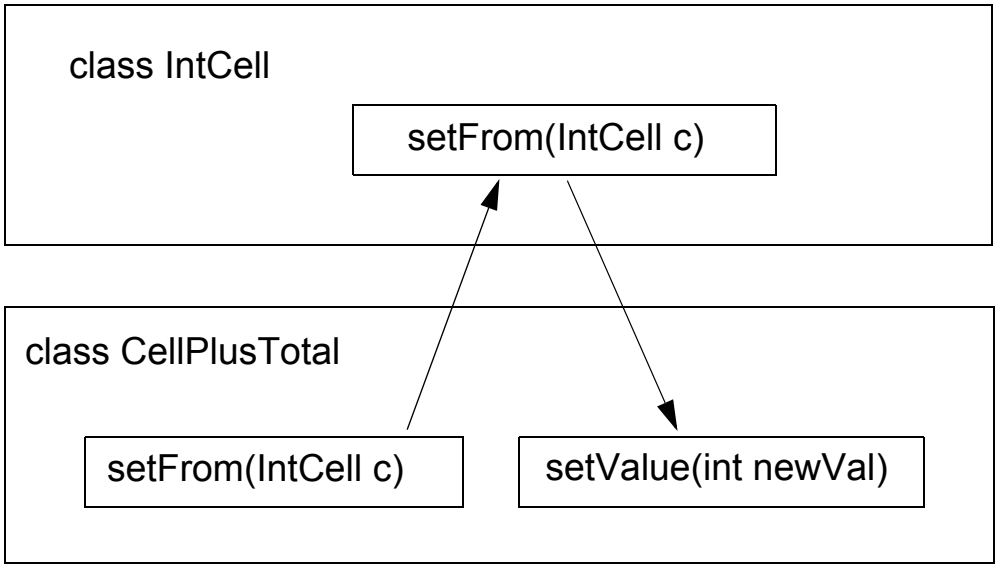

Figure 1.9: Diagram showing the downcall introduced by the override of methods `setFrom` and `setValue` in class `CellPlusTotal` based on the code shown in Figures 1.7 and 1.8.

Figure 1.6 gives the protected specification of `CellPlusTotal`. Figure 1.7 shows a possible implementation of `CellPlusTotal` based on its public and protected specifications and those of its superclass `IntCell`. This seems like a reasonable implementation since the superclass method is called to satisfy the inherited specification and then `_totalChg` is updated to satisfy the additional subclass specification. However, this implementation may not work correctly depending on the implementation of the superclass methods. Consider the implementation of method `setFrom` of class `IntCell` given in Figure 1.8. This superclass method calls methods `setValue` and `equals`. Because `setValue` is overridden, when `super.setFrom` is invoked by subclass methods, a downcall is made to `setValue` as shown in Figure 1.9. Because of this downcall to `setValue`, `_totalChg` will sometimes be updated twice. Therefore, without the code of the superclass and because of possible downcalls, it is not possible to reason about the state of the new instance variable `_totalChg` unless the documentation includes more information about the implementation of the superclass. Chapter 2 shows how such downcall problems can be avoided through a set of formal rules that specify when a superclass method must be overridden and when it can safely be called. These rules use the methods called and variables accessed by superclass and subclass methods in order to avoid these downcall problems. In JML, the calling structure is given in the callable clause of the subclassing contract, and the variables directly accessed is given in the accessible clause.

1.4.3 The Subclassing Contract

A primary purpose of the subclassing contract is to give programmers additional information they can use to avoid problems like those caused by downcalls. The tool described in Chapter 6 automatically generates the subclassing contract for the concrete methods of a class. Because constructors can also be involved in downcall problems, the subclassing contract is also generated for constructors.

The *subclassing contract* is composed of the callable and accessible clauses that occur in method specifications (see Figure 1.10). These clauses specify properties of the method's implementation and provide the additional information needed to safely create correct subclasses without superclass code. The callable clause lists the signatures of methods (and constructors) that can be called by the method being specified; and the accessible clause lists the variables whose value is accessed (read).

Also, the subclassing contract will normally be part of the protected specification since its primary users will be customizers and not clients of the class, even though some specification cases containing accessible and callable clauses may be public (Chapter 4 will explain why these clauses may have to be public). The subclassing contracts in Figure 1.10 show the callable and accessible clauses for class `IntCell`. In particular, Figure 1.10 shows the clauses for methods `setFrom` and `setValue` that would automatically be derived from the code given in Figure 1.8 (see Chapters 4 and 6 for more details).

1.4.4 Summary

The above example shows one of the ways that downcalls can be problematic when reasoning about the behavior of subclasses. Chapter 2 provides an analysis of other kinds of problems that can arise due to downcalls, and it describes a formal system for reasoning about how to create correct subclasses using only the specification of the superclass. Our technique demonstrates how the potential problems caused by downcalls can be avoided using the information specified in the subclassing contract. In particular, our technique avoids downcall problems through a set of rules that use the subclassing contract's specification of the methods called and variables accessed by superclass methods.

Downcall problems are closely related to the so-called *semantic fragile base class problem* [MS98] [SGM02, pp. 115-117], which is concerned with how to change superclasses without invalidating existing subclasses. The problem being addressed in this thesis might be called the *semantic fragile subclassing problem*, because it deals with how to create a valid subclass; that is, how to make super-calls and override superclass methods in such a way that the subclass and its superclass methods are free from problems caused by downcalls.

```
//@ refines "IntCell.java";

public class IntCell {
  /*@ also
    @ public code normal_behavior
    @    accessible initVal;
    @    callable \nothing;     @*/
  public IntCell(int initVal);

  /*@ also
    @ public code normal_behavior
    @    accessible newVal;
    @    callable \nothing;     @*/
  public void setValue(int newVal);

  /*@ also
    @ public code normal_behavior
    @    accessible c, c.value;
    @    callable this.equals(IntCell), this.setValue(int);     @*/
  public void setFrom(IntCell c);

  /*@ also
    @ public code normal_behavior
    @    accessible c, IntCell.value, this.value;
    @    callable IntCell.getValue();     @*/
  public /*@ pure @*/ boolean equals(Object c);

  /*@ also
    @ public code normal_behavior
    @    accessible this.value;
    @    callable \nothing;     @*/
  public /*@ pure @*/ int getValue();

  /*@ also
    @ protected code normal_behavior
    @    requires c != null;
    @    accessible c.value;
    @    callable c.getValue();     @*/
  protected IntCell(IntCell c);
}
```

Figure 1.10: `IntCell`'s subclassing contract from `IntCell.refines-java`.

1.5 Protecting Internal Objects

Reasoning from specifications can be unsound without some method for controlling or preventing unwanted aliasing and side-effects. This is because the state of an object *x* can change unexpectedly when some other external object makes changes to an aliased internal component of *x*. When this happens the state of *x* appears to be changing on its own[8]. As the following example will show, such modifications can make it difficult to maintain the integrity and correctness of internal objects or to maintain invariant properties of an abstraction. By definition an object is *internal* to object *o* if it can be referenced through one of *o*'s instance variables or indirectly through the fields of one of *o*'s other internal objects.

Internal objects are not protected from aliasing or representation exposure by most object-oriented programming languages. Scope rules and visibility modifiers, like private and protected in Java and C++, encapsulate and restrict access to the names of instance variables, but not the objects referenced by these names. Furthermore, in Java, an object can access private instance variables in other objects of the same class or protected instance variables in the same package. Therefore, the documentation of a class should have enough information so programmers can control and prevent the unwanted aliasing of its internal objects. The example in Figures 1.11-1.13 shows some of the kinds of problems that can arise when internal objects are aliased. Figure 1.11 gives a formal public specification of class `DeltaCell` and Figure 1.12 gives its protected specification. The JML keyword invariant introduces a class invariant; this assertion must hold on entry and exit from all public methods. Therefore, the invariant is implicitly conjoined with the pre- and postconditions of public methods[9].

Aliasing becomes an issue when an object's abstract value is implemented in terms of the fields of other objects, i.e., when its abstract value is determined from the state of one or more of its internal objects. This is the case in Figure 1.12, that is, `DeltaCell` is implemented using an `IntCell` object referenced by field `_delta`. Furthermore, the represents clause says that the abstract value of `delta` is determined from the `value` model field of `_delta`. Therefore, we need to specify that `_delta.value` is a member of the `delta` data group since any change in one will change the other; this is done through the maps clause given in Figure 1.12. The maps clause is used to specify that a field referenced indirectly (e.g. `value`) via the declared field (`_delta`) is a member of a data group (`delta`) [Lei98, LPHZ02].

If the class `DeltaCell` is implemented as in Figure 1.13, then aliasing of `_delta` is possible, even probable. Such an implementation behaves properly as long as any aliased internal `IntCell` objects are never modified by other objects; however, this is not guaranteed. Therefore, as will be seen

8. The state of an internal object is only important if it determines or is part of the abstract value (i.e. abstract model) of *x*; this will be explained and formalized in Chapter 3.

9. See Chapter 2 for more details.

```
public class DeltaCell extends CellPlusTotal {

   //@ public model int delta;
   //@                    in value, totalChg;

   //@ protected invariant delta == Math.abs(totalChg - value);

  /*@  public normal_behavior
    @    requires delta != null;
    @    assignable value, totalChg, delta;
    @    ensures value == initVal && totalChg == 0
    @        && this.delta == Math.abs(initVal);
    @*/
  public DeltaCell(int initVal, IntCell delta);

  /*@  public normal_behavior
    @    ensures \result.value == delta;
    @*/
  public /*@ pure @*/ IntCell getDeltaCell();
}
```

Figure 1.11: `DeltaCell`'s public specification from file `CellPlusTotal.jml-refined`.

```
//@ refines "DeltaCell.jml-refined";

public class DeltaCell extends CellPlusTotal {

    protected IntCell _delta;
   /*@                    in delta;
     @                    maps _delta.value \into delta;
     @*/

   //@ protected represents delta <- _delta.value;

   //@ protected constraint \old(_delta.value) <= _delta.value;

   //@ protected invariant _delta != null;
}
```

Figure 1.12: `DeltaCell`'s protected specification in file `DeltaCell.jml`.

below, reasoning about the correctness of such an implementation requires knowing where every actual and potential alias might occur in any client program (which is not modular and in general is not possible). The difficulties caused by the combination of mutable objects and aliasing are not restricted to reasoning about superclasses; they can also make reasoning about the correctness of subclasses difficult.

Figure 1.14 is a client program of classes `IntCell` and `DeltaCell`. The numbered comments following some statements shows the trace of the state of the objects `i1`, `d1`, and `d2` respectively. The state of `DeltaCell` objects are represented by a triple where the first element is the value of `_val`, the second the value of `_totalChg`, and the third the value of `_delta.value`. Consider the last statement of method `main`, the statement before the comment line numbered (9). Reasoning about whether or not this statement changes the state of `d1` depends on whether or not method `d1.getDeltaCell` exports `d1._delta` as its result; nothing about this can be mentioned in the formal public specification of `getDeltaCell` in Figure 1.11 since protected variables are not visible in public specifications. However, even more disconcerting is the fact that the state of `i1` appears to be changing on its own. Furthermore, when reasoning from public specifications, in the object `d1`, the assertion "`Math.abs(value - totalChg)`" will no longer have the same value as `_delta.value`, in violation of the invariant clause. Furthermore, because of aliasing, the invariant no longer holds at comment lines (3), (7), and (9). Also, at lines (2), (4), (5), (8), and (9), the value of `i1` is changing contrary to the specification of the method called in the statement immediately preceding that comment. Hence, standard correctness proofs using public pre- and postconditions and class invariants will be unsound unless specifications provide the information necessary so one can reason about and prevent unwanted aliasing. This also shows that reasoning about the program will require the code of the superclasses `IntCell` and `CellPlusTotal` unless specifications have sufficient information about the aliasing properties of methods.

Aliasing of mutable objects makes understanding even a small, simple program like the one in Figure 1.13 difficult. This example also shows that aliasing can, in a sense, create the same kinds of reasoning problems that are caused by dynamic scope. In dynamic scope, the object to which a parameter refers depends on the order of execution of procedure and function calls. Similarly, the number of aliases and where they occur depends on the control flow of a program. This simple example shows why reasoning can become difficult when the same statement may modify different objects in the system depending on which object is being referenced; this in turn, like dynamic scope, may depend on the statements previously executed by the program, including previous method calls.

Therefore, some way of controlling aliasing and side-effects to internal objects is necessary to make modular, useful, and sound static reasoning about the behavior of superclass methods possible. Furthermore, correct subclasses cannot be created without superclass code unless this information is available to reusers and programmers extending these classes. The alternative is to consider all

```
//@ refines "DeltaCell.jml";

public class DeltaCell extends CellPlusTotal {

  protected IntCell _delta;

  public DeltaCell(int initVal, IntCell delta) {
      super(initVal);
      _delta = delta;      // creates an alias
      _delta.setValue(Math.abs(initVal));
  }
  public void setFrom(IntCell c) {
      setValue(c.getValue());
  }
  public void setValue(int newVal) {
      super.setValue(newVal);
      _delta.setValue(Math.abs(_totalChg - _val));
  }
  public /*@ pure @*/ IntCell getDeltaCell() {
      return _delta; // exposes the internal representation to aliasing
  }
}
```

Figure 1.13: A fragment of `DeltaCell's` implementation from the file `DeltaCell.java`.

possible patterns of behavior of the class; this requires knowledge of all possible subclasses and client programs which in general is not possible for extensible class libraries.

Our general approach to this problem is to specify when methods are allowed to create aliases and when methods that modify the state of an existing object can be called. Furthermore, our technique provides a set of rules that, if followed, allow some kinds of aliasing while preventing aliasing could allow unexpected side-effects, that is, aliasing that disallows sound reasoning about method behavior such as that shown in the above example. Chapter 3 describes the specification technique and rules that control aliasing and side-effects. Also, tool support that can be used to enforce the restrictions necessary to make the reasoning system sound and practical is described in Chapter 6.

1.6 Approach, Contributions, and Assumptions

1.6.1 Approach

Our general approach to safely creating correct subclasses without seeing the superclass code has been to analyze and categorize the ways that problems can arise when reasoning about the behavior of subclasses, in particular, the ways that downcalls and aliasing can cause problems. When a problem is

```
public class Main
{
    protected static IntCell i1;
    protected static DeltaCell d1;
    protected static DeltaCell d2;

    public static int main () {
        i1 = new IntCell(3);
        d1 = new DeltaCell(1, new IntCell(5));
                                // (1) i1=3, d1=(1,0,1)

        // after the next statement, field d2._delta and i1 are aliases

        d2 = new DeltaCell(2, i1);
                                // (2) i1=2, d1=(1,0,1),  d2=(2,0,2)
        i1.setValue(4);
                                // (3) i1=4, d1=(1,0,1),  d2=(2,0,4)
        d2.setValue(6);
                                // (4) i1=2, d1=(1,0,1),  d2=(6,4,2)
        d2.setFrom(d1);
                                // (5) i1=8, d1=(1,0,1),  d2=(1,9,8)
        i1 = d1.getDeltaCell();
                                // (6) i1=1, d1=(1,0,1),  d2=(1,9,8)
        i1.setValue(5);
                                // (7) i1=5, d1=(1,0,5),  d2=(1,9,8)
        d1.setValue(4);
                                // (8) i1=1, d1=(4,3,1),  d2=(1,9,8)
        d1.getDeltaCell().setValue(2);
                                // (9) i1=2, d1=(4,3,2),  d2=(1,9,8)
    }
}
```

Figure 1.14: A program that uses classes `IntCell` and `DeltaCell` to illustrate the difficulties created by aliasing.

discovered, information is added to the specification when necessary and a formal system is developed with rules to prevent such problems. For example, a correct superclass method can only be problematic when a new subclass overrides one or more methods, leading to downcalls. Chapters 2 and 3 describe the formal systems and rules for disallowing the problems caused by downcalls and aliasing. In a few cases, when there is no easy way to soundly prevent the problem, we give restrictions on superclass code that would eliminate the problem (see assumptions below).

To understand what information is needed by programmers, we have studied a formal version of this problem. Our formal specifications for superclasses represent the documentation of a class library

or framework. Our reasoning technique corresponds to using the documentation. The ability to prove correctness of the subclass is used as a criteria to judge whether these specifications and the reasoning technique are adequate. Chapter 4 provides a proof of the soundness of the rules and technique. Therefore, we believe our study also provides sound guidance for providing adequate information in user manuals and informal documentation.

In our study, formal public and protected specifications were created for a base class which was then implemented. Based on this implementation, a subclassing contract was derived by hand, simulating what our tool would do. We next gave formal public and protected specifications for several new subclasses and studied the problem of how to correctly implement them without access to the superclass code.

The rules presented in Chapters 2 and 3 generalize our experience and provide a formal system for avoiding downcall and aliasing problems. The rules are conservative, because we are assuming that superclass code is not available, and thus the rules can only use information from specifications. The rules allow a programmer to determine which methods to override (Chapter 2), when it is safe to call a superclass method (Chapter 2), and when it is safe to call methods with side-effects (Chapter 3).

1.6.2 The Three Part Specification

To allow the safe creation of a subclass without using the source code of its superclasses, our approach uses a three-part specification. The above example shows that the documentation and specifications of classes in a library or framework should at least include the following three components:

1. a public specification that uses a client-visible model of objects of the class to describe the behavior of each public method and constructor,
2. a protected specification that describes any additional subclass-visible behavior, as well as the behavior of protected methods and constructors, and
3. a subclassing contract that specifies the methods that can be called and the variables that can be accessed by each public and protected method and constructor.

A complete specification is formed from the public and protected specifications, together with the subclassing contract. The complete specification is what programmers would use to implement subclasses. (JML also comes with a tool that can automatically combine specifications from various files into a complete specification [RL03, BCC+03]).

1.6.3 Class Library and Framework Implementation Guidelines

Reasoning about how to override superclass methods in such a way that downcall problems are avoided can become somewhat complicated [RL00]. Thus, to make the technique more practical, another contribution of this dissertation is the development of a set of guidelines for library

implementers that significantly simplifies the reasoning of programmers who are inheriting from library classes. Chapter 5 describes these guidelines.

1.6.4 Tool Support

Manually detecting the problems that may arise due to downcalls and aliasing can also become complicated and difficult. The tools described in Chapter 6 show how customizers, when specializing or extending library classes, can be assisted in the application of our technique. The rules form the basis of our tools which automatically generate the subclassing contract and give error messages when the rules are violated by the subclass.

1.6.5 Notation and Terminology

The application of our rules (as given in Chapters 2 and 3) need to distinguish the various kinds of calls, because the effects on downcall and aliasing problems can sometimes be subtly different, due to the different semantics of each kind of call. The definitions are given here because they are also needed in the assumptions given in the next subsection.

A *self-call* is a call on the current receiver object, i.e., the object denoted by `this` in Java and C++. For example, `setValue(u)` is a self-call, which is syntactic sugar for `this.setValue(u)` in Java and `this->setValue(u)` in C++.

A *super-call* is also a call on the current receiver object except that such calls invoke methods of a superclass. In Java, the built-in variable `super` is used to indicate a super-call, e.g., `super.setValue(u)`. In C++, super-calls are indicated by qualifying the name of the method, i.e., by giving both the superclass and the name of the method being invoked, e.g., `IntCell` **::** `setValue(u)`.

In our technique, a superclass constructor call must also be included in our definition of super-call because it can be involved in the same kinds of downcall problems for the same reasons, and thus the same rules apply to both. A *superclass constructor call* is an invocation of a superclass constructor by a constructor of the subclass. For example, `super(i)` is a superclass constructor call in Java. In Java these calls may only appear as the first statement in the body of a subclass constructor. In C++, superclass constructor calls invoke the constructor of a specific, named superclass; they can only appear in the member initializer list of a constructor declaration, e.g., a constructor invocation such as `IntCell(i)` could appear in the initializer list of one of `IntCell`'s subclass constructor declarations.

In contrast, an *object-call* is a call on an object other than the current receiver; e.g., `o.setValue(u)` is an object-call in Java or C++. Another kind of call is the *new object constructor call*; it is an expression, such as `new IntCell(5)` in Java or C++, that creates and initializes a new object by invoking a constructor.

Subclass methods and constructors can call superclass methods and constructors through super-calls. In the other direction, superclass methods and constructors can call subclass methods via downcalls. Therefore, when necessary to avoid possible ambiguities, methods and constructors will

sometimes be represented as *C*::*M*, where *M* is the method or constructor and *C* is the class in which *M* is defined. This is the same notation used in C++.

1.6.6 Assumptions

Our investigation focuses primarily on the issues involved in code inheritance and subclassing in a single-dispatch[10] OO programming language. Other language-specific features such as nested classes and exceptions are not considered. We, therefore, make several simplifying assumptions about the constructs allowed in our core OO programming and specification languages.

A1: We assume that types do not declare inner types or anonymous types.

A2: We assume that methods do not declare or throw exceptions.

A3: We assume that types do not declare static variables or methods.

A4: We assume that types do not declare package (default) visible variables or methods.

A5: We assume that types do not declare array variables.

We do not include exceptions in our core Language because this feature can be simulated by other constructs in Java. To further simplify our language, we do not include static variables and methods (in the Java sense), even though we believe that they can be handled as variables and methods of special class objects, as in Smalltalk; we leave this extension as future work.

Although not included, we also believe that our technique can be extended to handle package visible methods and fields. One approach would be to handle them in the same way as protected members when they are visible to a particular subclass, i.e., by making a package visible specification available to customizers of such subclasses. However, these members are not visible to all subclasses, i.e., when the superclass and subclass are not in the same package; in this case, package visible methods and fields have to be handled in the same way as private methods and variables so that, when necessary, subclasses know that they exist (see subsections 2.6 and 3.6.2). We leave the details as future work.

We further believe that the technique given in Chapter 3 for protecting internal objects from unexpected side-effects can be extended to handle arrays. However, arrays would have to be handled as a special case since every array element is publicly visible. For example, array objects can be simulated by an object that has one public field for each array element, but our technique assumes that the concrete fields of an object are not public and can only be modified via an object-call (see assumption A7, below). Thus if any array element is accessed by an invariant or represents clause, then the state of that entire array object must be protected from unexpected side-effects. Furthermore, we would need special rules to protect array elements when they are objects; we leave this and the handling of objects with public fields as future work (see also Chapter 3).

10. In *single dispatch*, the method to be executed is selected based on the dynamic type of the receiver. Smalltalk, C++, Eiffel, and Java use single dispatch.

Assumptions A1-A5 are the main simplifying syntactic assumptions we make. Chapter 4 gives the subset of Java and JML used in the formal soundness proof of our technique. However, the soundness of our approach also requires that we make several additional assumptions about the programming style of programmers and specifiers when they implement and document extensible types in a class library. Assumption A6 disallows code that should rarely, if ever, occur in a Java program, and certainly not when the superclass code is unavailable (explained in subsection 2.9.4.2). Assumption A7 is included to simplify the soundness proof of our verification logic given in Chapter 4. The assumptions, A8-A9, will be enforced by our tool since they are necessary for soundness. They restrict the way side-effects on objects other than the receiver can be implemented in a method[11]; these restrictions disallow assignments to concrete instance variables that could invalidate the run-time type invariant. We use the term *instance variable* for what Java calls a non-static data field.

A6: We assume that no super-calls are made before a superclass constructor has initialized the superclass fields (see subsection 2.9.4.2).

A7: We assume that all represents clauses specify an abstraction function with a consistent, well defined meaning [LM06], e.g., there are no cyclic or mutually recursive constraints among multiple model fields (see subsection 4.6.2).

A8: We assume that methods do not assign to instance variables of objects other than the current receiver object, e.g., `this` in Java and C++ (see subsection 2.4.3). To prevent most such assignments, we further assume that concrete instance variables do not have public visibility.

A9: We assume that private methods with side-effects are only directly invoked via self-calls; that is, they are not directly invoked via object-calls (see subsection 2.4.3).

The next two assumptions are restrictions that either the specifier or the specification language itself must enforce so our technique can be applied, i.e., so customizers (and our tool) have the information needed to apply our rules. For example, assumption A11 would not always be possible without assumption A10 since assumption A10 permits private fields to be referenced in the protected specification used by customizers; these two assumptions allow our technique (and our tool) to properly handle fields that otherwise would not be visible in the subclass.

A10: We assume that all private concrete fields are declared with the JML spec_protected modifier (see subsection 2.6.1). This modifier allows such fields to have private visibility in Java but protected visibility in JML specifications.

A11: We assume that all class invariants[12] are declared with public or protected visibility (or spec_protected) so they are visible to customizers (see subsections 2.6.2 and 3.3.2).

---

11. The rules in Chapter 3 require additional restrictions on assignment statements to control aliasing.

12. The represents clause specifies an invariant relationship between concrete variables and a model field. Therefore, both the invariant and represents clauses must be visible to customizers.

The next two assumptions, if followed, cause fewer subclasses to be unimplementable when the superclass code is not available (see subsection 2.9.3).

A12: We assume that if a class is extensible (e.g., is non-final in Java), then all of its methods visible to clients and customizers (e.g., public and protected in Java) are overrideable (e.g., are virtual in C++ or non-final in Java) if they have side-effects (see subsections 2.9.2 and 2.9.3).

A13: We assume that classes do not have methods that are mutually recursive with methods from unrelated classes (see Section 2.5). An *unrelated class* of *C* means a class other than *C* or one of *C*'s ancestors or descendents, i.e., outside the class hierarchy of *C*.

The last two assumptions below are only required if a concrete data refinement is necessary when implementing a subclass (see subsection 2.7); they are extensions of the closely related assumptions A8 and A12.

A14: We assume that methods do not access (read) instance variables, even public ones, of objects other than the current receiver object (see Section 2.7).

A15: We assume that if a class is extensible, then all of its non-private methods are overrideable if they access (read) instance variables of the receiver (see Section 2.7).

Chapters 2 and 3 give examples and further explain why the above assumptions are necessary and why overrideable methods allow for more implementable subclasses (see subsection 2.9.3).

1.7 Outline of Dissertation

Chapter 2 describes a formal system showing how potential downcall problems can be avoided using only the behavioral specifications and subclassing contract of superclass methods. The need for the rules and how they are applied will be illustrated using various examples; their soundness will be proven in Chapter 4.

Because aliasing can have surprising effects on assertions about the state of a system, the soundness and modularity of any verification system depends on the ability to control and prevent unwanted aliasing and side-effects. Chapter 3 provides a set of rules and a specification technique for preventing the kind of unwanted aliasing that would otherwise undermine the soundness of our reasoning technique.

Chapter 4 introduces the subset of Java and JML that is used in the soundness proof; it includes the core features of OO languages but omits some features that are not essential. Features are not essential if they can be simulated by other constructs. For example, exceptions can be simulated using other features of Java and will be excluded from the core language. Also features such as threads will be excluded to avoid the added complexities of parallelism. Chapter 4 also gives the axiomatic semantics of a core subset of JML.

A proof of the soundness of the rules and technique described in Chapters 2 and 3 is given in Chapter 5. The proof uses and is based on the core Java and JML languages introduced in Chapter 4.

The soundness of the technique is based on the ability to verify correctness of the code of the new subclass with respect to its specification. This is done using the formal specifications of both the subclass and its superclasses, and the code of the subclass, but without any superclass code.

Chapter 6 describes a set of guidelines for library implementers and reusers based on the rules given in Chapters 2 and 3. It shows how the library provider can simplify reasoning of reusers inheriting from the library if both adhere to these guidelines. In particular, it shows that the guidelines can prevent some of the downcall problems from ever arising and thus the associated rules would never have to be applied. Chapter 6 also briefly describes a tool that would automatically generate subclassing contracts and assist in applying the rules given in Chapters 2 and 3. Chapter 7 discusses related work, and offers some conclusions.

CHAPTER 2: PREVENTING UNVERIFIABLE BEHAVIOR

2.1 Introduction

Programmers need some way of reasoning about how to avoid the kinds of downcall problems that can arise when creating subclasses especially if the superclass code is not available. This chapter gives a set of rules for dealing with these problems. The rules only use information in specifications, i.e., in particular, they use the method behavior as specified in the public and protected specifications and the subclassing contract.

An important requirement of our technique is that it be statically checkable through an automated tool. Therefore, our technique is more conservative than might otherwise be necessary. For example, if our technique included rules that required handcrafted proofs or other human assistance, then we could, in some cases, allow subclass implementations that are currently disallowed (see subsection 2.9.4 for examples). However, we wanted to keep our technique and rules as simple as possible, statically checkable, yet not so restrictive as to be impractical.

Furthermore, a primary goal of our tool is that it warn when the subclass implementation may not be verifiable without the superclass code. Therefore, the tool will warn that a superclass method has to be overridden when its behavior has been extended (refined) or when the superclass method may no longer satisfy its specification. However, it may be that this method only has to be reverified (which is not done because the tool assumes the superclass code is unavailable and it does not do verification). Nonetheless, we believe that our tool and technique will be useful even when the superclass code is available, because it warns of potential problems that all customizers of inherited superclasses must reason about and handle. Furthermore, if the tool does not warn of potential problems, i.e., all our rules have been followed, then the subclass implementation can be verified without superclass code and none of the super-called or unoverridden superclass methods have to be reverified (formalized in Chapter 4).

The rules also assume that methods do not create unexpected side-effects. For example, if the value of a field is directly or indirectly part of an object's abstract value, then such fields can only be modified during the execution of one of that object's methods (formalized in Chapter 3). Such fields are components of the representation and must not be exposed to objects in an unknown context where they could be modified unexpectedly. Therefore, we assume these fields are not publicly visible and the objects that contain them are not aliased. Chapter 3 describes how our technique controls and prevents such unwanted aliasing and representation exposure.

There are two reasons that a superclass method will have to be overridden by a new subclass. First, a superclass method has to be overridden if the subclass needs to extend (refine) its behavior. Second, it must be overridden if it might no longer satisfy its specification or there are aspects of its behavior that cannot be verified. Furthermore, a superclass method must not be super-called by subclass methods if it may no longer satisfy its specification or it makes unexpected changes to subclass fields.

Therefore, our rules fall into two main categories. There are rules for determining which methods must be overridden and rules for determining when super-calls are unsafe. All rules are conservative because we want them to be checkable by an automated tool and because we assume the superclass's code is unavailable.

The *overriding rules* determine which methods must be overridden. This set of rules must be applied repeatedly until no additional methods are added to the set of methods to be overridden. Downcall problems for the subclass, other than those caused by super-calls, are avoided when the overriding rules are followed and our assumptions (see subsection 1.6.6) have also been followed.

The *invalidation rules* specify when it is unsafe to make a super-call. These rules are closely related to the overriding rules but their purpose is to prevent callback problems involving super-calls. The rules state the conditions under which a superclass method cannot be called because it may no longer satisfy its specification or may modify subclass variables in unverifiable ways; that is, they specify when a superclass method has been invalidated by the new subclass. A superclass method is *invalidated by a new subclass* if, because of downcalls, it might no longer satisfy its superclass specification or it might cause side-effects to subclass variables that cannot be verified without the superclass code.

Some special rules related to the handling of private variables and super-calls are explained in Sections 2.6 and 2.8. For example, there are situations where super-calls are mandatory; thus a special rule is needed to handle such cases, e.g., when superclass methods modify and maintain private instance variables.

The sections describing our rules are followed by a discussion of some consequences. This discussion and a summary appear in Section 2.9.

2.2 New Subclass Instance Variables

When creating subclasses a programmer must be able to reason about the effects that methods have on the subclass’s new instance variables. Our technique uses data groups (defined in subsection 1.4.1) to control which subclass variables can be modified by an overriding subclass method. If the superclass code is not available, then reasoning about the state of subclass instance variables can be problematic; these problems happen when a superclass method makes downcalls and there is a data group dependency (defined below) between superclass and subclass variables. This section gives examples that illustrate the kinds of reasoning problems that can result when superclass and subclass fields have a dependency relationship and the superclass code is unavailable. This section also gives a set of rules for avoiding and preventing these problems, i.e., our rules ensure that one can reason soundly about the behavior of a newly created subclass.

2.2.1 Additional Side-Effects

We start by describing the main concepts and specification techniques used by our rules. In particular, we introduce the concept of additional side-effects and provide more details about the use of data groups and JML's in, maps, and assignable clauses.

2.2.1.1 Data groups

Our technique uses data groups together with the assignable clause to specify and control side-effects. Recall that a *data group* represents a set of fields grouped together because they are related in some way and tend to change as a group (see Section 1.4). In JML, the declaration of each model instance field automatically creates an associated data group with the same name as the field[1]. A field and its implicit data group always have the same visibility level. For example, in Figure 2.2, data group `oldVal` has public visibility, the same as its associated model field. Furthermore, a field is always a member of its associated data group.

A model field and its data group are used to hide a subset of the underlying concrete instance variables of a class. For example, model fields are usually public, but concrete fields are protected or private so unrelated classes cannot directly access or assign to these concrete fields. When a field with public or protected visibility, such as a model field, is listed in a method's assignable clause, then this denotes the associated data group. When a data group is listed in a method's assignable clause, then that method can assign or call methods that assign to any of the concrete fields in that group. However, our technique only allows methods to directly assign to fields of the receiver object. That is, each instance variable can only be directly assigned by methods of the object whose type or supertype contains a direct declaration of that variable. An instance variable `x` is *directly declared* in type *T* if `x` is declared as one of *T*'s data fields. Field `x.V` is *indirectly declared* in type *T* if `x` is directly declared in *T* and `x` references an object with a directly or indirectly declared field *V*. Furthermore, we say that a field is *declared* in type *T* if it is either directly or indirectly declared in *T*. For example, in Figure 1.12, `_delta` and `_delta.value` are both declared in class `DeltaCell`. However, `_delta` is directly declared and `_delta.value` is indirectly declared. Therefore, our restrictions on assignment allow methods of `DeltaCell` to directly assign to `_delta`, but `_delta.value` can only be modified through object-calls on `_delta`[2].

2.2.1.2 Data group dependencies

Adding a field to a data group through an in or maps clause declares a *data group dependency*. Thus a field has a *dependency relationship* if it is a member of another field's data group. We say that data group membership specifies a dependency relationship because, for example, a model field

1. Concrete instance fields can also have an associated data group when they have a spec_public or spec_protected modifier (see Chapter 4).
2. Subsection 2.4.3 explains why this restriction is necessary in our technique; it is one of the assumptions given in subsection 1.6.6.

depends on the value of members of its data group. That is, a model field derives its abstract value from some or all of the concrete fields in its data group as in Figures 1.5, 1.6, and 1.12.

Data group dependencies can be *static* or *dynamic* [LS99, LN00, LPHZ02]. In JML, the in clause declares a *static dependency* and the maps clause declares a *dynamic dependency*. Figure 2.1 illustrates the distinction; that is, the in clause of class `Point` declares a static dependency because it relates a data group to a field from the same object, i.e., `oth` and `x` are members of `this` (the receiver). In contrast, the maps clause adds an indirectly declared variable to a data group; it declares a dynamic dependency between a group and field from different objects. For example, the maps clause in class `Point` of Figure 2.1 relates a group and field from different objects, i.e., the indirectly declared field `oth.x` is added to the data group `this.x`. We say the maps clause declares a dynamic dependency because an indirectly declared field like `oth.x` also changes whenever the object referenced by `oth` changes. The maps clause involving `_delta.value` in Figure 1.12 is another example of the declaration of a dynamic dependency[3].

2.2.1.3 Visibility requirements

For modular soundness, the declaration of a data group dependency must be visible everywhere that the field being added to a data group is visible [Lei98, LN00, LN02, LPHZ02]. For example, a static dependency, like the one declared by the in clause of class `Point` in Figure 2.1, must be visible everywhere that variable `oth` is visible. Similarly, the maps clause and the indirectly declared field it adds to a data group must both have the same visibility. For example, the maps clause in class `Point` must be visible everywhere that `oth.x` is visible (i.e., everywhere that `oth` is visible)[4].

This visibility requirement is enforced through JML's syntax and type checker. For example, the syntax requires that in clauses immediately follow the declaration of the field being added to data groups. Also, the maps clauses must immediately follow the indirectly declared fields being added to data groups, i.e., JML requires that the maps clause reference one of the indirectly declared fields from the preceding field declaration. For example, the first maps clause of Figure 2.1 is allowed because it references the indirectly declared field `oth.x`, one of the fields of `oth`. However, the two maps clauses in subclass `BadPoint` are not allowed in JML because `oth.y` and `this.y` are not being declared there; that is, the dependency relationships declared by those maps clauses are not visible in the superclass where `oth.y` and `this.y` are declared and visible.

To understand why this visibility restriction is required, consider again the two maps clauses in subclass `BadPoint` of Figure 2.1. If these maps clauses were permitted, then an overriding subclass method like `BadPoint::setX` would be allowed to modify additional superclass variables, i.e., `y` and

3. A class level (i.e., static in Java) or global variable can also be a member of the data group of an instance variable but we do not consider those relationships here; we leave that for future work.

4. A dynamic dependency like this one can have problems if `oth` is aliased in an unknown context (see the example in Section 1.5). Chapter 3 describes how our technique prevents such unwanted aliasing.

```
public class Point {

  //@ public model int x, y;

    protected Point  oth;
    //@              in x;
    //@              maps oth.x \into x;

  /*@ public normal_behavior
    @   assignable x;
    @   ensures x == newX && oth.x == \old(x);
    @*/
    public void setX(int newX);
}

public class BadPoint extends Point {

  /*@ public model int z;
    @                  in oth.x;              // illegal
    @                  maps oth.y \into x;    // illegal
    @                  maps this.y \into x;   // illegal
    @*/

  /*@ also
    @ public normal_behavior
    @   assignable x;
    @   ensures y == newX && oth.y == \old(y);
    @*/
    public void setX(int newX);
}
```

Figure 2.1: Example of the correct and incorrect use of the maps clause.

`oth.y` because they would now be members of `x`'s data group. However, assignments to `y` and `oth.y` cannot be allowed since they are not permitted by the superclass specification of `Point::setX` in the class where `y` and `oth.y` are declared. Furthermore, if such assignments were allowed, `BadPoint` would no longer be a behavioral subtype of `Point` according to `Point`'s specification (see also Section 2.9.1 and the definition of behavioral subtype in Figure 1.1). Therefore, the two maps clauses in `BadPoint` cannot be allowed [Lei95, Lei98, LPHZ02, MPH01].

To summarize, a field and its dependency relationships are all declared together and, therefore, as is required for soundness, within the same context and scope. Also, the data group (in and maps) clauses have the same visibility level as the field being declared. For example, public is the visibility of the data group clause in Figure 2.2, and protected for the one in Figure 2.3. Furthermore, JML type

```
public class CellPlusPrevious extends IntCell {

  //@ public model int oldVal;
  //@                    in value;

  /*@  public normal_behavior
    @     assignable value, oldVal;
    @     ensures value == initVal && oldVal == initVal;   @*/
  public CellPlusPrevious(int initVal);

  /*@ also
    @  public normal_behavior
    @     assignable value;
    @     ensures oldVal == \old(value);   @*/
  public void setValue(int newVal);

  /*@ also
    @  public normal_behavior
    @     requires c != null;
    @     assignable value, oldVal;
    @     ensures oldVal == \old(value);   @*/
  public void setFrom(IntCell c);

  /*@  public normal_behavior
    @     requires c != null;
    @     assignable value, oldVal;
    @     ensures value == c.value && oldVal == \old(value)
    @         && \result == Math.abs(value - oldVal);    @*/
  public int copyFrom(IntCell c);

  /*@  public normal_behavior
    @     ensures \result == Math.abs(value - oldVal);   @*/
  public /*@ pure @*/ int getChange();

  /*@  public normal_behavior
    @     requires c != null;
    @     ensures \result == Math.abs(this.value - c.value);   @*/
   public /*@ pure @*/ int difference(IntCell c);
}
```

Figure 2.2: Public specification of `CellPlusPrevious` from file `CellPlusPrevious.jml-refined`.

```
//@ refines "CellPlusPrevious.jml-refined";

public class CellPlusPrevious extends IntCell {

    protected int _prevValue;
    //@                 in oldVal;

    //@ protected represents oldVal <- _prevValue;
}
```

Figure 2.3: Protected specification of `CellPlusPrevious` from the file `CellPlusPrevious.jml`.

checking does not allow a field to be a member of a data group that is less visible than it is since the data group would not be in scope at the higher visibility level. For example, a public field cannot be a member of a private or protected data group.

Finally, the syntax also requires that the group referenced by an in or maps clause be a member of the current receiver, i.e., a field cannot be added to an indirectly declared data group[5]. That is, a field cannot be added to a group from an object other than the receiver; thus the only qualified data group names allowed by the syntax of the in and maps clause are those with prefixes "`this.`" and "`super.`" since these prefixes denote members of the current receiver. For example, the in clause of `BadPoint` is not allowed because data group `oth.x` is from an object other than the receiver[6].

Nonetheless, data groups can contain fields that are members of different objects. For example, in class `DeltaCell` of Figure 1.12, `_delta` and `_delta.value` are members of the same data group. However, fields declared in the receiver, such as `this._val`, cannot be members of a data group declared in another object, such as `_delta.value`, since the syntax of the in and maps clauses only allow fields to be added to a data group declared in the current receiver; this restriction is necessary for soundness.

2.2.1.4 Nested data groups and indirect dependencies

Our technique also allows data groups to be nested. That is, if field *V* is a member of data group *X*, then *V*'s data group will always be a subset of data group *X*. Nesting of data groups is necessary for declaring indirect dependencies, e.g., the dependency between *X* and, by transitivity, members of *V*'s data group. This is also a convenience because, otherwise, declaring this dependency would require

5. For simplicity we do not consider static methods, fields, or data groups in this dissertation; however, in JML, static fields can be added to a static data group that is not a member of a receiver object. We leave the details for handling static members as future work.

6. There are also other syntactic restrictions required for alias control; this is described in Chapter 3.

that *V*'s members also be explicitly added to data group *X*[7]. Notice, however, that the members of data group *V* may be assignable in methods in which the other members of *X* are not assignable.

A typical example of data group nesting and indirect dependencies is given in Figures 2.2 and 2.3. Figure 2.2 shows the public specification of `CellPlusPrevious`, a new subclass of `IntCell`. This subclass adds three new methods, `copyFrom`, `getChange`, and `difference`, and extends the specifications of `setValue` and `setFrom`; the also keyword indicates that a method specification is being extended. The protected specification is given in Figure 2.3. This new subclass adds a new concrete instance variable, `_prevValue`, that stores the previous value of the cell object. The dependency relationship declared by the in clause of Figure 2.2 nests the subclass data group `oldVal` inside `val`ue from the superclass; this allows model field `oldVal` to be updated whenever `value` is changed. The in clause of Figure 2.3 also allows the concrete variable `_prevValue` to be assigned whenever methods can modify `oldVal`. This second in clause creates an indirect data group relationship between `value` and `_prevValue`; that is, by transitivity, `_prevValue` can also change when `value` changes.

2.2.1.5 Additional side-effects

A superclass method can have additional side-effects if the subclass specification allows that method to assign to fields in addition to those allowed by the superclass specification. This is permitted when the subclass specification inserts additional fields into a data group that the superclass method can already modify. However, for soundness, a subclass is not allowed to add superclass fields to a data group (as explained in subsection 2.2.1.3). Therefore, a superclass method can only have additional side-effects if the subclass specification allows that method to assign to a new subclass variable[8].

More formally, a method *M* can have *additional side-effects on W*, if *S* is a subclass of *C*, *M* is declared in superclass *C*, *W* is directly or indirectly declared in subclass *S*, and *W* is assignable by *C*::*M*. To permit additional side-effects on *W*, *W* has to be added to a superclass data group. That is, since the superclass knows nothing about potential subclass fields, the subclass specification can only permit these additional side-effects by adding *W* to a superclass data group that is modifiable by *C*::*M*. For example, in subclass `CellPlusPrevious` of Figure 2.2, a superclass method can have additional side-effects if it is allowed to modify `oldVal` (or implicitly the members of its data group). Thus methods `setValue` and `setFrom` are allowed to have additional side-effects because they can modify `value` which contains `oldVal`; however, `copyFrom` does not have additional side-effects because it is not defined in the superclass. Note also that field `oldVal` is superfluous in the assignable clause of `setFrom` because `oldVal` is a member of `value`'s data group, i.e., both `value` and `oldVal` are

7. Such nesting is also used by the rules in our technique for alias control given in Chapter 3.

8. Note that this includes both directly and indirectly declared subclass fields. Also, the preconditions in each specification case must also be considered when determining which fields can be assigned (see Chapter 4 for more details).

assignable any time `value` is listed. Thus the assignable clause of `setValue` is allowed to omit `oldVal` without changing the meaning (although often it may be clearer for clients if superfluous fields are listed).

Also, additional side-effects cannot be implemented directly in a superclass method because subclass variables are not in scope in the superclass. That is, additional side-effects can only be implemented through a method override, i.e. an implementation in the subclass, or through downcalls to methods that assign to the subclass variable.

When is it necessary to specify that a method has additional side-effects, i.e., when is it necessary to add a subclass variable to a superclass data group? In general, a subclass is required to specify that a method *M* can have additional side-effects on *W* if:

1. *M*'s postcondition implies an assertion *A(W)* involving a new subclass variable *W*, and
2. *M* needs to be able to modify *W* in order to ensure that *A(W)* holds on exit from *M*.

M's postcondition may refer to *W* either explicitly or implicitly; that is, it may explicitly reference *W* or it may implicitly reference *W* through a variable *V* when the values of *W* and *V* are related through either a represents clause or an invariant. For example, in Figure 2.2, the public postconditions of `setValue` and `setFrom` explicitly reference `oldVal`; however, they implicitly reference the concrete variable `_prevValue`, through the in and represents clauses in Figure 2.3.

When an overriding method *M* can modify a subclass variable *W*, our approach requires both a data group clause and an assertion, *A(W)*, implied by *M*'s postcondition. For example, if the superclass specification allows *M* to modify a variable *V* and the subclass specification says *M* must also modify *W* in order to achieve some effect related to *V*, then *W* must be added to *V*'s data group, i.e., a data group clause relating *V* and *W* is required [Lei95, Lei98, LPHZ02]. For example, in Figure 2.2, the specification of method `setValue` requires it to modify `oldVal` using the previous state of superclass variable `value`. Therefore, the in clause shown in Figure 2.2 is needed to allow this. Furthermore, the in clause shown in Figure 2.3 is also necessary, otherwise the JML type checker will emit an error whenever an assignment to `_prevValue` occurs in the implementation of `setValue` and `setFrom`.

On the other hand, if there is a data group clause in the subclass that allows *M* to modify *W*, then *M*'s postcondition needs to imply an assertion *A(W)* about the state of *W*; otherwise, if there is no assertion, then *W* can take on an arbitrary value allowed by any applicable represents and invariant clauses. Thus, another implicit assumption is that the subclass and its implementer care about the state of *W* (although perhaps our assumption is too strong for those rare situations when the state of a subclass instance variable does not matter).

2.2.2 The Additional Side-Effects Overriding Rule

This subsection describes examples of both explicit and implicit specifications that require the implementation of additional side-effects by a subclass. It explains how our rules handle these additional side-effects through method overrides.

Additional side-effects are often required when an assertion *A(W)* in the subclass specification relates a superclass field *V* to a subclass field *W* since changes in one variable may require changes to the other. Therefore, without superclass code or more information, it cannot be proven that a superclass method *M* that modifies *V* will make the correct changes to *W* so that the subclass specification is satisfied. That is, either *M*'s postcondition, as given in the subclass, or the subclass invariant may not be satisfied unless *M* is overridden.

For example, the assignable clauses in Figure 1.2 for methods `setValue` and `setFrom` together with the data group clauses of Figures 2.2 and 2.3 specify that variables `value`, `oldVal`, and `_prevValue` are assignable in these methods. The value of `_prevValue` is modified as specified in the ensures clause of these methods, taking into account the represents clause in Figure 2.3. (This represents clause says that the value of `oldVal` is given by the value of `_prevValue`.) The specification of subclass `CellPlusPrevious`, says that all methods that change the cell's value will also have to store the previous value of the cell in `_prevValue`. The rule below requires method overrides so this can be done.

Additional Side-Effects Overriding Rule. Let *S* be a subclass of *C*. If *S* specifies that method *C*::*M* can have additional side-effects on field *W* or if method *C*::*M* makes a self-call down to a method that may have additional side-effects on *W*, then *C*::*M* must be overridden.

This rule requires a method override if *C*::*M* can assign to a subclass field *W*. To specify that *C*::*M* can have additional side-effects on *W*, *W* must be added to a superclass data group that is assignable by *C*::*M*. Furthermore, the specification of *S*::*M* must not prohibit assignment to *W*. In JML, assignment to *W* can be prohibited by adding the assertion \not_assigned(*W*) to *S*::*M*'s postcondition. Additional side-effects are necessary when *M*'s postcondition, in the subclass specification, implies an assertion *A(W)* about the state of *W*. Thus the override is needed to ensure that *A(W)* holds on exit from *M* since we assume that the superclass code is unavailable.

When *W* is added to a superclass data group, this may also allow method *C*::*M* to modify *W*. However, an override can be avoided, if *C*::*M* does not make downcalls to methods that assign to *W* and *S*::*M* prohibits assignment to *W*. Therefore, when additional side-effects are not needed in *M*, an override of *M* can sometimes be avoided in our technique by explicitly specifying that *M* does not change *W*[9]; this situation usually occurs when additional side-effects are needed in some method other than *M*.

Notice, however, that when *C*::*M* makes a downcall to a method that can assign to *W*, this rule requires an override because we assume the superclass code is unavailable. That is, even if *S* does not allow *C*::*M* to assign to *W*, we cannot verify, without the code (or more information), that such

9. The specification would have to be such that \not_assigned(*W*) applies to the postconditions of all specification cases of *S*::*M*.

downcalls do not assign to *W*. Therefore, we have to assume that such downcalls may violate *C*::*M*'s specification regarding side-effects to *W*[10].

Furthermore, when a method override is necessary because of additional side-effects on *W*, the subclass needs to specify the behavior of *M* with respect to *W*, i.e., how to change *W*. That is, the assertion *A(W)* about how *M* modifies *W* must be available in the specification of subclass *S*. This can be done either explicitly in the postcondition of *S*::*M* or implicitly through a represents clause or invariant. For example, in Figure 2.2 the assertions regarding the model variable `oldVal` are explicit in the method specifications. However, due to subclass invariants and represents clauses, there may be an implied assertion that does not appear explicitly as a conjunct in the postcondition. For example, in Figure 2.2 the postconditions do not explicitly specify how `_prevValue` is to be changed by these methods. However, this is specified indirectly by the represents clause shown in Figure 2.3, that is, `_prevValue` must have the same value as `oldVal`. In some cases, when invariants are involved (see discussion of Figure 2.4 in subsection 2.2.2.2 below), it is not as obvious as in Figure 2.2 that a superclass method needs to be overridden. However, the rule tells us to inspect the superclass specification and to identify the methods with additional side-effects any time a subclass field is added to a superclass data group. The reason *C*::*M* has to be overridden is to make sure *W* is properly modified and the specification is needed for this.

The specification of *M*'s behavior is also necessary for the verification of the implementation of *S* and its subclasses (whether they are concrete or abstract). Therefore, even if a subclass like `CellPlusPrevious` were abstract, the behavior of every method allowed to modify subclass fields (i.e., `oldVal` and `_prevValue`) must be given in the specification (as shown in Figures 2.2 and 2.3) so verifiers know how these methods modify these subclass fields.

2.2.2.1 Abstract classes

The above rule states that *C*::*M* has to be overridden, but it does not necessarily require that method *M* be overridden in *S*. In particular, when *S* is abstract, it may not be necessary (or possible) for *S* itself to implement the overriding method *M*. For example, *W* may be a model field and *M* may be an abstract method. Thus the behavior of *M* could be specified in terms of *W*, but the represents clause for *W* and the concrete fields used to compute the abstract value of *W* may not be specified in *S*. Therefore, the concrete fields, the represents clause for *W*, and the overriding implementation of *M* would have to be defined in a subclass of *S*. That is, the implementation of the additional side-effects on *W* would have to be done in concrete subclasses of *C*, i.e., in *S* or in the concrete subclasses of *S* by overriding or inheriting an override of *M* with the appropriate additional side-effects on *W*.

Similarly, the above rule could also be applied to interfaces. For example, when an interface *S* extends another interface *C*, then only the specification can be provided in *S* since all interface

10. JML's checking of the assignable clause does not catch this error because it does not consider \not_assigned expressions in the postconditions of methods.

methods are abstract. The override occurs in the concrete class that implements *S*. We will not consider interfaces further in this dissertation because they do not involve downcalls or callbacks and all methods have to be overridden (implemented). However, the above rule can be used to determine when a specification is required by an overriding interface method, i.e., an extending specification is required when a method has additional side-effects (see also Section 2.3).

In summary, the above rule specifies when *C*::*M* has to be overridden and, implicitly, when *S*::*M* must provide a specification that has to be satisfied by this overriding method. However, the rule does not require that an abstract subclass *S* override *M*, only that *C*::*M* be overridden by *S* or one of *S*'s concrete subclasses.

2.2.2.2 Subclass invariants

Another similar problem that occurs in the presence of subclass instance variables is related to subclass invariants. In JML, the keyword invariant introduces properties that must hold in all publicly visible states of objects of the class. If more than one invariant clause occurs in a class, then all the given properties must hold in such states. Together, the conjunction of these properties is the *invariant* for the class[11]. For example, subclass `CellPlusInvariant`, specified in Figures 2.4 and 2.5, has a class invariant; the invariant, shown in Figure 2.5, specifies that the difference between the previous cell value and the current value is to be stored in `_diff`.

In JML each invariant clause has a visibility level, e.g., public, protected, or private. The class invariant is the conjunction of all invariant clauses at all visibility levels.

The class invariant must hold at the beginning and end of all methods; it must also hold on exit from all constructors[12]. Therefore, when verifying an implementation of a method, the class invariant will be conjoined with the method's pre- and postconditions. It is also conjoined with the postcondition of all constructors. The class invariant is also conjoined with the precondition of Java finalizers, but does not have to hold in their post-state. Similarly in C++, the class invariant is conjoined with the precondition of destructors.

For example, the invariant in Figure 2.5 is implicitly conjoined with the postconditions of all methods and constructors declared in class `CellPlusInvariant`. This invariant specifies that the value of concrete variable `_diff` is determined by `value` and `oldVal`. Thus `_diff` must be allowed to change whenever `value` or `oldVal` change; this is done using the in clause, as shown in the protected specification of Figure 2.5. Furthermore, the in clause and invariant clauses implicitly extend the behavior of all methods that change variables `value` or `oldVal`. That is, they implicitly

11. JML also includes history constraints that, like invariants, have to be preserved by all methods; however, for simplicity, they will not be considered here.

12. JML has a special modifier, helper, for private methods that are not required to establish the class invariant. Not requiring that a private method establish the class invariant is only sound because methods of other classes cannot call it. In such cases, if private methods could be called by other classes, then the class invariant may not always hold prior to a later method call that expects it.

```
public class CellPlusInvariant extends CellPlusPrevious {

  /*@ public normal_behavior
    @   assignable value, oldVal;
    @   ensures value == initVal && oldVal == initVal;
    @*/
  public CellPlusInvariant(int initVal);
}
```

Figure 2.4: `CellPlusInvariant`'s public specification, from `CellPlusInvariant.jml-refined`.

```
//@ refines "CellPlusInvariant.jml-refined";

public class CellPlusInvariant extends CellPlusPrevious {

   protected int _diff;
   //@            in value, oldVal;

   //@ protected invariant _diff == Math.abs(value - oldVal);
}
```

Figure 2.5: `CellPlusInvariant`'s protected specification, from `CellPlusInvariant.jml`.

introduce additional side-effects into the postcondition of methods `setValue` and `setFrom` because the invariant must hold on exit from these methods.

Therefore, when invariant relationships are specified in the subclass, the required behavior of subclass methods can be implicit. That is, subclasses like `CellPlusInvariant` are not necessarily required to explicitly specify the behavior of overriding methods because the extending specifications are formed by conjoining the subclass invariant with the pre- and postconditions of the inherited specifications. Hence, the subclass invariant specifies how subclass fields (e.g., `_diff`) are to be modified by overriding methods with additional side-effects, as in Figure 2.5.

However, without superclass code, it is not possible to know whether the subclass invariant has been established prior to making a downcall to a non-private[13] method. For example, a superclass method, *M*, could modify variables that invalidate a subclass invariant; if this occurs, it is unsafe for *M*

13. Private methods are not involved in the downcall problems described here because they cannot be called or overridden by other classes. See subsection 2.6.4 to see how private methods are handled in our technique.

to self-call down to a subclass method since it requires and assumes that this invariant holds. Consider, for example, the implementations of classes `CellPlusPrevious` and `CellPlusInvariant` given in Figures 2.6 and 2.7 respectively. Method `copyFrom` of `CellPlusPrevious` calls down to method `getChange` of `CellPlusInvariant` which expects the subclass invariant given in Figure 2.5 to hold. Since this invariant involving `_diff` may not hold, the return value may no longer satisfy `copyFrom`'s superclass specification, i.e., it will always return the old value of `_diff`.

However, such problems involving subclass invariants will often be prevented by the required method overrides of the above Additional Side-Effects Overriding Rule. That is, the notion of data groups (from subsection 2.2.1) can be used as a way of conservatively detecting which superclass and subclass variables might be related by an invariant. For example, if assignment to a superclass variable requires changing a subclass variable in order to satisfy a subclass invariant (as in `CellPlusInvariant` of Figure 2.5), then these variables have to be members of the same superclass data group[14]. Furthermore, when an inherited superclass method *M* can modify the fields in this data group, then *M* can have additional side-effects. Therefore, relationships between variables, as in `CellPlusInvariant` of Figure 2.5, require an override based on the Additional Side-Effects Overriding Rule; also, the overriding method must ensure that the subclass invariant holds on exit.

Furthermore, even though the subclass invariant is conjoined with the post-condition of every instance method, this does not necessarily require that all superclass methods be overridden. That is, we only have to require an override when it is necessary to make sure the subclass invariant holds on exit from a superclass method. Therefore, a subclass invariant only requires an override if the superclass method has side-effects that could change or invalidate the invariant, i.e., when it has additional side-effects.

2.2.3 The Additional Side-Effects Invalidation Rule

The concept of additional side-effects has another important purpose as related to super-calls. Superclass methods can have unexpected behavior if they have unexpected additional side-effects. Our first invalidation rule prevents subclass methods from making super-calls to methods with unexpected or unverifiable additional side-effects. We say that a superclass method is *invalidated by a subclass* if the superclass method may no longer satisfy its superclass specification or its behavior with respect to subclass fields is unverifiable without the superclass code.

Since subclass variables are not visible in the superclass, superclass methods can only have additional side-effects if they make downcalls to methods that assign to subclass variables. For example, consider again the Java implementation in Figure 1.7; the Additional Side-Effects Overriding Rule above says that `CellPlusTotal` must override the two methods shown. Calling the superclass method to satisfy the inherited superclass specification and then updating `_totalChg` seems like an

14.Chapter 3 also requires these data group relationships as part of our technique for alias control.

```
public class CellPlusPrevious extends IntCell {

  protected int _prevValue;

  ...
  public int copyFrom(IntCell c) {
      _prevValue = _val;
      _val = c.getValue();
      return getChange(); // expects any class invariants to hold
  }
}
```

Figure 2.6: A fragment of `CellPlusPrevious`'s implementation from the file `CellPlusPrevious.java`.

```
public class CellPlusInvariant extends CellPlusPrevious {

  protected int _diff;

  public CellPlusInvariant(int initVal) {
      super(initVal);
      _diff = 0;
  }

  ...
  public int getChange() {
      return _diff;
  }
  public int copyFrom(IntCell c) {
      _diff = super.copyFrom(c);  // incorrect because of downcalls
      return _diff;
  }
}
```

Figure 2.7: A fragment of `CellPlusInvariant`'s implementation from the file `CellPlusInvariant.java`.

obvious implementation of these methods. However, these implementations will only be correct if we can guarantee that the subclass fields are not changed during the super-calls. This cannot be guaranteed because the superclass implementation of method `setFrom` shown in Figure 1.8 makes a downcall to `setValue`; this may introduce unexpected side-effects to variable `_totalChg` that result in incorrect behavior, i.e., `_totalChg` may be updated twice. Hence, such super-calls cannot safely be reasoned

about without superclass code (or more information) because they may call down to a method with additional side-effects[15]. To prevent such super-calls, we introduce the following invalidation rule.

Additional Side-Effects Invalidation Rule. Let *S* be a subclass of *C*. Let *S* specify that superclass method (or constructor) *C*::*M* can have additional side-effects on field *W*. If *C*::*M* self-calls down to a method *S*::*N* that is allowed to modify *W*, then *C*::*M* may not be super-called by any method (or constructor) of *S*.

The above rule says that a superclass method is invalidated and cannot be called by the subclass, if it directly or indirectly makes a downcall to a method that may have additional side-effects (this can be determined from *C*::*M*'s subclassing contract). For example, the incorrect implementation of `setFrom`, shown in Figure 1.7, would not be allowed because it makes a super-call to an invalidated method. That is, because of the downcall to method `setValue` (that may modify `_totalChg`), the method `setFrom` in Figure 1.8 cannot be super-called by methods in `CellPlusTotal` of Figure 1.7. (The downcall would be indicated in the callable clause of `IntCell` **::** `setFrom`.) Therefore, we say that `IntCell` **::** `setFrom` *has been invalidated* by the subclass `CellPlusTotal` based on the Additional Side-Effects Invalidation Rule.

On the other hand, the super-call in the implementation of `setValue`, shown in Figure 1.7, would be permitted by this rule, since `IntCell` **::** `setValue` has not been invalidated by the new subclass; that is, it does not make any downcalls.

In the statement of the above rule, *M* and *N* may refer to the same method, e.g., when *M* is recursive; thus a method with additional side-effects is always invalidated by the subclass if it is recursive. Also, the rule includes constructors since superclass constructors can be invoked by subclass constructors, and when invoked from a subclass, they can have the same kinds of callback problems as methods[16].

However, constructors cannot be overridden and the type of the object being initialized by a constructor is determined statically, so, for constructors, the only problems that have to be avoided are those introduced by super-calls. Therefore, constructors are not mentioned in the overriding rules but are included in the invalidation rules, when applicable.

In summary, we have given examples that show that anytime a superclass method *M* has additional side-effects, then it may no longer satisfy its superclass specification or may have unverifiable behavior (without superclass code) if it makes downcalls. That is, if *M* makes downcalls to methods with additional side-effects, then, without superclass code, it is not possible to prove that these modified subclass variables will have the correct value after the call. Furthermore, if *M* modifies a superclass field that is constrained through a subclass invariant clause, then it may not be safe to make

15. In Section 2.8, we introduce an authorization rule that allows such super-calls if specific restrictions can be followed.

16. Constructors can be viewed as methods invoked to initialize an object.

downcalls to methods that, on entry, may require that the subclass invariant hold. That is, we illustrated with an example that, without superclass code (or more information), one may not be able to prove that the subclass invariant holds prior to a downcall. Therefore, the above Additional Side-Effects Invalidation Rule is needed so these superclass methods will not be super-called.

2.3 Method Refinement

In the previous section we discussed the ramifications of adding new subclass variables and how our technique controls and allows for the implementation of additional side-effects. However, additional side-effects are not the only reason that a subclass specification may require a method override. A method override is also required whenever the subclass refines (extends) the behavior of a superclass method, even if there are no additional side-effects. In this section we describe a more general overriding rule. This rule overlaps with the Additional Side-Effects Overriding Rule but does not replace it.

In particular, additional side-effects are not the only side-effects that could require an override. For example, superclass methods that assign to any of the fields accessed by a subclass invariant may have to be overridden to make sure the subclass invariant is not invalidated (our technique requires an override since we assume the superclass code is unavailable[17]). Our next rule handles these more general cases involving side-effects that could affect the validity of the subclass invariant.

In addition, a method override may be required when a subclass explicitly extends the behavior of a superclass method. For example, the superclass specifications of methods `setValue` and `setFrom` are explicitly extended in Figure 2.2; the also in those specifications indicates that the superclass specifications are being extended, i.e., refined. Therefore, in cases where the extended specifications are explicit as in Figure 2.2, it should be obvious that an override is required. The override is needed so the extended subclass specifications can be satisfied by the subclass implementation. Our next overriding rule handles these cases where the subclass refines the superclass specification.

Method Refinement Overriding Rule. Let *S* be a subclass of *C*. If the specification of *S*::*M* refines the behavior of superclass method *C*::*M*, then *C*::*M* must be overridden.

A subclass specification refines the behavior of a superclass method M if *M* can have side-effects on fields referenced by a subclass invariant clause or if *S*::*M* has requires and/or ensures clauses that extend *M*'s inherited behavior. Also, an ensures clause is not considered a refining specification if its only assertion is a \not_assigned expression that prohibits additional side-effects (since the purpose of such expressions is to avoid an override). Therefore, based on this definition, the behavior of method *C*::*M* can be refined either by an explicit subclass specification, as in Figure 2.2, or through a subclass invariant, as in Figure 2.4.

17. If the superclass code is available, then the superclass method would have to be reverified or overridden to ensure the subclass invariant holds.

Furthermore, we have defined refinement of method behavior so it can be used and applied statically by our tool. Thus our tool will determine method refinement based on whether any of a superclass method's assignable fields are accessed by subclass invariant clauses[18] and whether a method's subclass specification has requires or ensures clauses.

The examples in Figures 2.2 and 2.4 also illustrate that the above rule overlaps with the Additional Side-Effects Overriding Rule. However, the Additional Side-Effects Overriding Rule is still needed to make sure a superclass method *M* is overridden if *M* makes downcalls to methods that may have additional side-effects; this is necessary because, without the superclass code (or more information), we cannot reason about *M*'s behavior with respect to the subclass fields modified by downcalls (also detecting additional side-effects is easier than analyzing the subclass invariant and represents clauses when determining whether an override is necessary, so this easier analysis can be done first by the tool).

Furthermore, the above method refinement rule is conservative because the rule (and the tool) do not allow for a proof that a superclass method's behavior does not invalidate the subclass invariant or require an override. For example, when the subclass has invariant clauses, if the customizer can prove that the postcondition of a superclass method implies the subclass invariant, then an override is not necessary. Such a proof is trivial when a method does not assign to fields referenced by subclass invariant clauses because such side-effects cannot change the invariant's validity. Therefore, the above rule does not require an override when a superclass method does not have side-effects that affect the subclass invariant. Also, an override may not always be necessary when our tool detects side-effects that are thought to affect the invariant. However, we do not believe this to be a problem with our technique because the customizer would still have to do a proof that the invariant holds using the superclass specification and code; thus this information is still useful to customizers because it warns of potential problems.

Furthermore, a superclass method will not necessarily be invalidated just because it had to be overridden due to a refinement in its behavior. That is, a superclass method should not be invalidated as long as it satisfies its superclass specification and does not assign to subclass fields; when this is true, the superclass specification can be used in verification and in reasoning about super-calls because super-calls are explicit and can be treated like static calls.

However, when refining the specification of a superclass method, the specifier should make sure the specifications do not conflict. That is, superclass method specifications are inherited by subclasses in JML, so the inherited specification should not conflict with the subclass specification since both have to be satisfied. Thus the combined specifications should not cause a method's postcondition to be false. For example, if the invariant in a subclass of `IntCell` requires that `value` always be positive,

18.Chapter 3 describes how our tool determines the fields accessed and the limitations of our technique.

then this invariant would conflict with the postcondition of method `setValue` in Figure 1.2; that is, the postcondition and invariant cannot both be satisfied when the parameter `newVal` is less than zero.

2.4 Subclass Invariants

This section describes how our technique handles problems that may arise when the subclass extends the superclass invariant, i.e., declares additional invariant clauses. In particular, we explain why our invalidation rule and some of our assumptions are needed to prevent invalidation of the subclass invariant and to prevent method calls while the subclass invariant does not hold. In subsection 2.4.1 we give an invalidation rule for preventing super-calls to methods that may make downcalls while the subclass invariant is invalid. In subsection 2.4.2 we give a corresponding overriding rule that makes sure self-calls and object-calls do not make downcalls while the subclass invariant does not hold. In subsections 2.4.3 and 2.4.4, we explain why some of our assumptions are necessary for the soundness of our technique, i.e., are necessary to prevent invalidation of subclass invariants.

2.4.1 The Invariant Invalidation Rule

The above Additional Side-Effects Invalidation Rule (subsection 2.2.3) prevents super-calls when that call may assign to a subclass field via downcalls; we do not allow such super-calls because, without the superclass code (or more information), it may not be possible to verify or reason about the state of the modified subclass fields. However, this invalidation rule is not strong enough to prevent all super-calls to methods that may no longer satisfy their superclass specification, e.g., those that make downcalls when the subclass invariant does not hold. Consider, for example, Figures 2.6 and 2.7. In the implementation of method `copyFrom` of Figure 2.6, method `getChange` is called. However, the super-call of `copyFrom` in Figure 2.7 cannot be allowed because the overriding `getChange` method expects the subclass invariant to hold, i.e., it directly accesses the new subclass variable `_diff`. So there is no way of verifying the behavior of superclass method `copyFrom` without superclass code (or more information) and thus it must not be called from methods of `CellPlusInvariant`.

However, this and other similar problems are not prevented by the above Additional Side-Effects Invalidation Rule. For example, the above rule does not invalidate superclass method `copyFrom` of Figure 2.6 because its only downcall is to `getChange` which does not modify `_diff`; in fact, super-calls to `copyFrom` do not modify any subclass fields. Therefore, the above invalidation rule does not disallow the incorrect implementation of `copyFrom` given in Figure 2.7.

We could, however, consider strengthening the Additional Side-Effects Invalidation Rule so all superclass methods with additional side-effects are invalidated if they also make downcalls. However, this is also insufficient to prevent all the problems involving subclass invariants. For example, the subclass invariant clauses may only reference and constrain superclass fields; so a superclass method that invalidates the subclass invariant may not have additional side-effects. That is, additional side-effects are not the only side-effects that could invalidate a subclass invariant. As explained above,

assignments to any of the fields accessed by the subclass invariant could invalidate the subclass invariant.

Furthermore, strengthening this invalidation rule, in this way, would disallow super-calls that should not be eliminated; that is, it would disallow some verifiably correct implementations because it would, for example, invalidate superclass methods that make a downcall even though there are no subclass invariant clauses (i.e., the subclass has the same invariant as the superclass).

In summary, super-calls are safe if they satisfy their superclass specification and one can reason about their side-effects on subclass variables. Thus a superclass method will satisfy its specification if it does not make downcalls when the subclass invariant does not hold and does not make downcalls that modify subclass fields[19]. Therefore, in addition to disallowing super-calls that modify subclass fields via downcalls, our technique must also ensure that the subclass invariant holds prior to a method call. Furthermore, a superclass method's side-effects (if any) cannot invalidate a subclass invariant since it cannot re-establish an invariant it knows nothing about. The next rule invalidates such superclass methods when there is a subclass invariant that constrains assignable superclass fields.

Invariant Invalidation Rule. Let *S* be a subclass of *C*. Let *V* be a concrete instance variable visible in *C*. Let *S* specify an invariant that accesses and constrains the value of *V*. If superclass method *C*::*M* can assign to *V* and it makes a self-call, then *C*::*M* may not be super-called by any method (or constructor) of *S*.

The above rule disallows the incorrect implementation of Figure 2.7; that is, methods allowed to assign to `value` or `oldVal` may invalidate the subclass invariant because the subclass invariant indirectly accesses concrete fields `_val` and `_prevValue` through `value` and `oldVal` respectively. Hence, methods that also make downcalls would be invalidated, e.g., method `copyFrom` would be invalidated by the downcall to `getChange`. We have to disallow such super-calls because, as in this example, the superclass method may no longer satisfy its specification.

Our tool and technique are conservative because they assume that the superclass invariant does not imply the subclass invariant; however, if the customizer can prove this, then that method would not have to be invalidated. Furthermore, this rule will invalidate a superclass method even though all of its downcalls are made at a point where the subclass invariant holds, e.g., prior to any of that method's side-effects. Therefore, if the superclass code is available, then one may be able to reverify the superclass code in the context of the new subclass; that is, one may be able to prove that the subclass invariant always holds whenever an overridden method is downcalled.

Our approach, however, is to require an override (the Method Refinement Overriding Rule in Section 2.3) and to disallow super-calls to such methods (the above Invariant Invalidation Rule). Nonetheless, our tool would still be useful to customizers when superclass code is available because it warns when reverification is necessary and when there are potential problems in the implementation.

19.This claim is formalized and proven in Chapter 4.

Our tool will enforce the Invariant Invalidation Rule by analyzing the assertions in the subclass invariant clauses to determine which superclass fields are accessed and by determining whether these fields are assignable, i.e., whether these fields are referenced in a method's assignable clause (see Chapter 3 for more details on the issues involved in determining the fields accessed in an invariant and how this would be done).

Note also that this rule has to prevent a superclass method from being called when it makes self-calls, not just downcalls. That is, because subclasses lower in the hierarchy than *S* may override the method being self-called, we have to consider all self-calls to be potential downcalls. Thus this rule must invalidate superclass methods that make self-calls when *S*'s invariant may have been invalidated.

2.4.2 The Invariant Overriding Rule

Downcalls can also occur during a self-call of an unoverridden superclass method. For example, methods `setValue` and `setFrom` in `CellPlusInvariant` (Figure 2.4) are allowed to modify `value` and `oldVal` which could invalidate the subclass invariant. Therefore, our technique must require an override when a superclass method assigns to fields that could invalidate the subclass invariant. This rule is similar to the Invariant Invalidation Rule except that we require an override to prevent self-calls to invalidated superclass methods.

Invariant Overriding Rule. Let *S* be a subclass of *C*. Let *V* be a concrete instance variable visible in *S*. Let *S* specify an invariant that accesses and constrains the value of *V*. If superclass method *C*::*M* can assign to *V*, then *C*::*M* must be overridden in *S*.

Clearly such methods will usually have to be overridden since the superclass knows nothing about subclass invariants. However, in general, it is possible that all super-class methods establish the subclass invariant; since this would require a proof, our technique assumes that assignments to superclass variables invalidate the subclass invariant.

2.4.3 Explicit Parameter Objects

The rules given so far deal with problems caused by side-effects to fields of the implicit receiver parameter (`this` in Java and C++). Our technique prevents these problems by overriding and invalidating superclass methods and constructors. We would like to generalize these rules to handle side-effects on explicit parameter objects. However, in a single dispatch language like Java we cannot override methods based on the types of a method's explicit arguments; they can only be overridden based on the type of the receiver object.[20]

Consider, as an example of these problems, Figure 2.8. A new method `setTo` has been added to the public specification of class `IntCell` in that Figure; this method modifies the state of its argument `c` by assigning its own `value` field to the `value` field of `c`. Figure 2.9 shows a possible

20. In the conclusion (Chapter 7), we consider the application of our rules to multiple dispatch languages, like MultiJava [CLCM00, Cli01], that allow overrides based on explicit argument types.

```
public class IntCell {

   //@ public model int value;       // model variable

  /*@  public normal_behavior
    @     assignable value;
    @     ensures value == initVal;   @*/
  public IntCell(int initVal);

...  // other methods are the same as in Figure 1.2

  /*@  public normal_behavior
    @     requires c != null;
    @     assignable c.value;
    @     ensures this.value == c.value;  @*/
   public void setTo(IntCell c);
}
```

Figure 2.8: Part of the public specification of `IntCell` in file `IntCell.jml-refined`.

```
public class IntCell {
  protected int _val;
  ...
  public void setTo(IntCell c) {
      c._val = _val;
   }
}
```

Figure 2.9: An incorrect implementation of method `setTo` for class `IntCell`. The code for setTo violates our assumptions, because it directly assigns to fields of another object.

implementation. However, `c.setTo` must also establish the subclass invariant when the receiver `c` has type `CellPlusInvariant`. Furthermore, when implementing method `setTo`, all classes must satisfy the specification given in Figure 2.10. This specification is implicit and comes into play as soon as there can be objects of type `CellPlusInvariant` in the program.

The implementation in Figure 2.9 creates problems because it directly assigns to a field of `c`, which may invalidate the subclass invariant in `c`. Even unrelated classes in the same package must satisfy this subclass invariant if they change the value of a cell object passed as an explicit argument. This example shows that, in a single dispatch language like Java, these kinds of problems cannot be eliminated by method overrides[21].

```
/*@ also
  @ implies_that
  @ protected normal_behavior
  @    requires c instanceof CellPlusInvariant;
  @    assignable c.value, ((CellPlusInvariant)c)._diff;
  @    ensures ((CellPlusInvariant)c)._diff
  @                    ==  Math.abs(c.value - \old(c.value));
  @*/
public void setTo(IntCell c);
```

Figure 2.10: Implicit specification of method `setTo` that must be satisfied by all classes because of the subclass invariant in `CellPlusInvariant`.

Furthermore, direct assignment to instance variables of any object, other than the receiver, can lead to similar problems that cannot be avoided by method overrides. Therefore, one cannot safely assign to fields of any object other than the receiver because that object may have unknown subclass invariants.

In addition, object-calls to an unoverrideable method with side-effects can also lead to similar problems. In fact, if a method *M* makes an object-call to an unoverrideable method *o.N*, then any assignments by *N* to an instance variable *V* will have the same effect as *M* making direct assignments to *o.V*, a field of an object other than the receiver. This is because *N* cannot be overridden so that it establishes a subclass invariant involving *V*[22].

In summary, if the type of an object is extensible, then the invariants of all possible subclass objects, including unknown new ones, must be established when modifying fields of that object. Thus direct assignment to fields, other than those of the receiver, are unsafe in general because some new subclass could cause those assignments to invalidate an invariant. Furthermore, such direct assignments are not modular, i.e., they require whole program knowledge of all subclasses that have an invariant involving subclass variables. In addition, all methods that directly assign to fields of an object (other than the receiver) might have to be rewritten in order to establish a new invariant anytime a new subclass is created and this subclass adds to the class invariant; this cannot be done for superclass methods if superclass code is unavailable.

Direct assignment to fields, other than those of the receiver, can also introduce unexpected behavior when subclass methods have additional side-effects. For example, consider again the implementation of method `setTo` given in Figure 2.9. When argument `c` has dynamic type

21. This problem is closely related to the well-known “binary method problem” [BCC+95].

22. However, this is also a problem for subclasses in general because, in such cases, the subclass would have a method that when called does not establish the subclass invariant. Therefore, in our technique, such subclasses are considered unimplementable (see subsection 2.9.3).

`CellPlusPrevious` (Figures 2.2 and 2.3), then `_prevValue` will not be updated with the previous value of the cell, which clients would probably expect. However, a (possibly) worse case occurs when `c` has dynamic type `CellPlusTotal` (Figures 1.4 and 1.6); in this situation, `c._totalChg` is not being updated when `c.value` is changed. Therefore, `_totalChg` will no longer reflect the total changes made to the cell's value since its instantiation. These two examples illustrate how some rather obscure bugs can be introduced when our restrictions on assignment are not followed.

Our technique avoids these problems by disallowing assignment to fields of objects other than the receiver, and by disallowing object-calls to unoverrideable methods (such as private methods) if they have side-effects (see assumptions in subsection 1.6.6). In fact, because of subclass invariants, our restrictions on assignment seem to be necessary unless the type of the object is unextensible[23]. Furthermore, from a software engineering perspective, these restrictions seem to be good practice because they simplify the reuse of superclasses. It is also our belief that non-private methods with side-effects should always be overrideable if they are members of an extensible class (see subsections 2.9.2 and 2.9.3).

Our restrictions on object-calls and assignment can certainly be incorporated into a programming method for creating large systems. This is illustrated by the existence of large Smalltalk systems; all methods in Smalltalk are overrideable and Smalltalk's scope rules do not allow methods to access or assign to fields of objects other than the receiver.

A correct implementation of `setTo` is given in Figure 2.11. In this implementation, `c.value` is modified by a call to the non-private method `setValue` and this method would have been overridden so that the subclass invariant is properly established. Therefore, our assumptions require that all side-effects to objects, other than the receiver, be accomplished indirectly through calls to overrideable methods that are required to establish the class invariant (e.g., calls to private methods are not allowed). In this way, the incorrect implementation given in Figure 2.9 is not allowed and an implementation like the one given in Figure 2.11 is required[24].

Another important point, however, must be considered when reasoning about methods that modify the state of argument objects. Our restrictions ensure that the subclass invariant is properly established, but a client can only safely reason using the specification of the type of the receiver object. For example, if `c` has type `CellPlusInvariant` and the receiver has type `IntCell`, then the client can only reason about the behavior of `setTo` using the specification of `IntCell` (Figure 2.8). That is, the client can reason about the value of `c.value` (since this is specified), but not about `c.oldVal` or

---

23. One could, however, consider allowing direct assignment to fields of objects that have an unextensible type, e.g., final classes in Java. However, all invariants and the fields involved would have to be appropriately visible in the context of the assignments so the invariants could be maintained.

24. One could "loosen" these restrictions in situations where the dynamic type of the object is known, but, in general, the type will not be known, and in particular, the type cannot be determined when explicit parameters are involved.

```
public class IntCell {
  protected int _val;
  ...
  public int setTo(IntCell c) {
      c.setValue(_val);
   }
}
```

Figure 2.11: A correct implementation of method `setTo` for class `IntCell`. The code for `setTo` no longer violates our assumptions, because it no longer directly assigns to fields of another object.

```
public class IntCell {
  protected int _val;
  ...
  public void setTo(IntCell c) {
      if (this.equals(c)) {
          return;
      } else if (_val < 0) {
          c.setValue(_val);
          c.setValue(_val);
      } else {
          c.setValue(_val);
      }
   }
}
```

Figure 2.12: Another implementation of method `setTo` for class `IntCell`.

`c._diff`. To understand why, consider the implementation of `setTo` given in Figure 2.12. This implementation satisfies `IntCell`'s specification (and our assignment restrictions), but variable `c.oldVal` may be left unchanged, or it could equal either the pre-state or post-state value of `c.value`, depending on how many times `c.setValue` is called. Therefore, without the superclass code, one can only reason in general from the specification of the static type of the receiver.

However, in the rare and unlikely case that 0, 1, or n calls to the methods listed in the callable clause all produce the correct (same) result, then one may be able to verify, without superclass code, that the superclass method making the object-call will assign the correct values to the subclass variables.

2.4.4 Temporary Side-Effects

A method has *temporary side-effects* if it changes an instance variable and then restores the original value before it returns. There are two kinds of problems that can arise when new instance

```
public class CellPlusPrevious extends IntCell {
  protected int _prevValue;
  ...
  public int setFrom(IntCell g) { ... }
  public int getChange() { ... }
  public int difference(IntCell c) {
      int saveVal = _val;
      int saveOld = _prevValue;
      setFrom(c);
      int d = getChange();
      _val = saveVal;
      _prevValue = saveOld;
      return d;
   }
}
```

Figure 2.13: An implementation of class `CellPlusPrevious`. The code for `difference` violates our assumptions, because it uses temporary side-effects.

variables are added to a subclass and a method has temporary side-effects. First, a problem can arise if a superclass method makes a downcall to a method with additional side-effects but the original value of the subclass field is never restored. For example, consider the implementation of `CellPlusPrevious` shown in Figure 2.13. In that figure, method `difference` is implemented using the `setFrom` and `getChange` methods. This implementation will not work properly when called from subclass `CellPlusInvariant`, because `setFrom` has additional side-effects that are not handled. That is, `_diff` will not be restored, in violation of the assignable clause of method `difference`. Furthermore, the invariant will no longer hold. Therefore, `difference` must be overridden and cannot be super-called by methods of `CellPlusTotal`.

A second problem can arise when a superclass method makes a downcall before the original value has been restored; this is illustrated by the implementation of subclass `CellPlusPrevious` given in Figure 2.14. In this figure only the method `getChange`, which has no additional side-effects, is used in the implementation of `difference`. However, even this implementation may not work correctly when called from a subclass like `CellPlusInvariant` of Figures 2.4 and 2.5, because the subclass invariant is not established before `getChange` is called. Since `difference` might not work correctly, it cannot be super-called by methods of `CellPlusInvariant` and would have to be overridden.

The first problem described is detected by JML's typechecker, that is, JML does not allow assignments to a subclass variable via downcalls unless the assignments are permitted by the method's assignable clause. But the second problem cannot be detected using only the specifications of a method's behavior because the downcall has no additional side-effects; that is, it cannot be determined that `difference` has to be overridden unless the variables that have been temporarily changed appear

```
public class CellPlusPrevious extends IntCell {
  protected int _prevValue;
  ...
  public int getChange() {
      return Math.abs(_val - _prevValue);
  }
  public int difference(IntCell c) {
      int saveOld = _prevValue;
      _prevValue = c.getValue();
      int d = getChange();
      _prevValue = saveOld;
      return d;
  }
}
```

Figure 2.14: Another implementation of class `CellPlusPrevious`, which also has temporary side-effects.

in the assignable clause. Therefore, JML does not allow temporary side-effects[25]; that is, in JML, variables may not be changed, even temporarily, by a method (or a method it calls) unless they are directly or indirectly (through a data group relationship) mentioned in the assignable clause. In this way, the additional side-effects overriding and invalidation rules can be used to prevent downcall problems even if variables are temporarily changed.

JML's semantics for the assignable clause is more restrictive than necessary since JML does not allow methods to temporarily change and then restore variables without including these variables in the assignable clause. That is, ignoring concurrency, it may only be necessary that temporary changes be disallowed around calls to non-private methods so the invariant holds when these methods are called. With this assumption, it suffices to interpret assignable clauses as only pertaining to modification of the value of variables between the pre- and post-states of a method; this less restrictive interpretation, however, is difficult to check statically, whereas the JML rule is easier to check statically.

2.4.5 Downcalls by Constructors

Some slightly different problems can occur when superclass constructors make downcalls. For example, the subclass invariant may not have been established prior to the downcall or, worse yet, the downcall may result in abnormal termination because a field contains a null pointer instead of a valid object reference. These problems can arise because subclass variables do not have to be initialized and

25. In part, JML also disallows temporary side-effects because they cause problems for reasoning about concurrent programs.

the type invariant does not have to hold prior to the invocation of a constructor. Our next rule disallows a superclass constructor call if it makes downcalls and the subclass contains an invariant clause.

Constructor Invariant Invalidation Rule. Let *S* be a subclass of *C*. If *S* specifies a subclass invariant and constructor *C*::*M* self-calls down to a method *S*::*N*, then *C*::*M* may not be super-called by constructors of *S*.

Notice that this rule is more conservative than the rule for methods (subsection 2.4.1) because, unlike constructors, the subclass invariant is established prior to the call of any superclass or subclass methods. Furthermore, a superclass method has to establish the superclass invariant prior to any downcalls so the only way a superclass method can invalidate the subclass invariant prior to a downcall is through its assignments to superclass fields. However, invariant relationships involving subclass variables will not change unless they are related to a modified superclass field. Therefore, a superclass method only has to be invalidated when its modification of superclass fields may invalidate the invariant specified in the subclass. Hence, the Invariant Invalidation Rule (subsection 2.4.1) only invalidates methods that make downcalls when a subclass invariant clause accesses one of the modifiable superclass fields.

Another related problem can occur when a superclass constructor makes a downcall to a method that accesses subclass variables before they have been properly initialized (subclass fields will not normally be initialized prior to a superclass constructor call). Furthermore, this problem may arise whether or not there is a subclass invariant clause; thus the next rule is necessary to prevent constructors from making downcalls before subclass fields have been initialized.

Constructor Initialization Invalidation Rule. Let *S* be a subclass of *C*. Let *W* be an instance variable directly or indirectly declared in *S*. If constructor *C*::*M* self-calls down to a method *S*::*N* that directly or indirectly accesses the value of *W*, then *C*::*M* may not be super-called by constructors of *S*.

This rule is conservative because it assumes subclass fields do not have valid values until after the subclass constructor has been executed; however, this can sometimes be avoided through some special coding tricks (see subsection 2.9.4.2).

Furthermore, in Java, the first statement of every constructor is a call to a superclass constructor; this is part of the semantics of Java. This constructor call can be explicit or implicit; the default constructor of the superclass is implicitly called when this statement is omitted in the subclass. Therefore, implementers of a subclass have to be careful when the superclass's default constructor is implicitly called. This implicit call has to be listed in the callable clause of the subclass constructor. Because of this required constructor super-call, a subclass will be unimplementable if all superclass constructors have been invalidated by the subclass.

2.5 Mutually Recursive Methods

So far we have described the way our technique handles additional side-effects when subclass fields are related to superclass fields. Now we want to explain how our technique handles termination issues.

Potential nontermination may arise when new methods insert themselves into cycles of mutually recursive methods. This section describes rules that make termination proofs possible when the code for mutually recursive superclass methods is not available. The purpose of the rules in this section is to prevent callback cycles involving both superclass and subclass methods.

A *callback cycle* occurs when a method, *M*, makes a call to another method, which then calls back to *M* (perhaps indirectly). A callback cycle means there is a cycle in the call graph, and thus there is potential for non-termination. The following rule prevents callback cycles in the call graph that would make a termination proof impossible without the superclass code or more detailed information[26].

Callback Cycle Overriding Rule. Let *P* be an overriding method in a subclass. If *P* self-calls a superclass method *M* that self-calls back directly or indirectly to *P*, then *M* must be overridden.

Because the above rule is applied repeatedly until all methods in the cycle have been overridden, the callback cycle rule ensures that all methods in a call graph cycle are overridden if any method in the cycle is overridden. The rule ensures that a callback cycle does not include both superclass and subclass methods. However, a callback cycle that occurs in the superclass will not occur in the subclass if the cycle is broken by one of the subclass methods. For example, consider Figure 2.15 in which the callable clauses show that method `search` may call `search`Left which may call `search`Right which then may call back to `search`. However, this cycle does not occur in the subclass, if `search` is overridden so it satisfies its specification, but this time without calling back to `search`Left or `search`Right. Therefore, when `search`Right calls down to `search`, there is no recursion; so `search` will terminate in a state satisfying its specification, and so will `search`Right. Thus, in this case, the callback cycle rule does not require that either `search`Left or `search`Right be overridden.

Another related problem occurs with mutually recursive callbacks among methods of different objects. However, our technique assumes, as stated in subsection 1.6.6, that groups of mutually recursive methods do not contain methods from unrelated classes. This seems to be needed for soundness because overriding all methods involved in such a cycle does not prevent nontermination,

---

26. Other features of a specification language, such as JML's measured_by clause, may make termination proofs possible in such cases, but these are outside the scope of this dissertation; our main purpose is to identify situations where a superclass and one of its subclasses interact such that unexpected behavior may arise. These situations are used to demonstrate what additional information needs to be provided so a correct subclass can be implemented even though superclass code is unavailable and method behavior is specified informally.

```
public class C {

  /*@
    @ protected code normal_behavior
    @    callable searchLeft(int );
    @*/
  public void search(int val);
}

  /*@
    @ protected code normal_behavior
    @    callable searchRight(int );
    @*/
  public void searchLeft(int val);

  /*@
    @ protected code normal_behavior
    @    callable search(int val);
    @*/
  public void searchRight(int val);
```

Figure 2.15: An example of a callback cycle involving methods `search`, `searchLeft`, and `searchRight`.

that is, overrides cannot prevent the callback cycle from including both superclass and subclass methods. Consider for example Figure 2.16. In that figure, suppose *A* and *C* are classes for which no code is available. Also, methods *A*::*M* and *C*::*N* are mutually recursive with *A*::*M* taking an argument of type *C*[27] and *C*::*N* taking an argument of type *A*. Figure 2.16 also shows subclasses *B* of *A* and *D* of *C* that override *M* in *B* and *N* in *D*. However, these overrides do not help the termination proof. For example, in `Main::main` the call to *B*::*M* passes an actual argument of type *C*, not one of type *D*. Therefore, *C*::*N* rather than *D*::*N* will be called; this could lead to nontermination because the superclass code for *C*::*N* is not available.

Thus our technique must prevent the kinds of nontermination problems described above. To accomplish this, our technique must invalidate those superclass methods that involve mutually recursive callbacks whenever termination cannot be proved. Therefore, our rules must always prevent a callback cycle that includes both superclass and subclass methods because we assume the superclass code is not available. For this reason, some super-calls cannot be allowed. Consider, for example, the implementation of `CellPlusPrevious` and `CellPlusInvariant` shown in Figures 2.17 and 2.18.

27.It could instead access a field of type *C* in one of its arguments.

```
public class A {

  /*@
    @ protected code normal_behavior
    @    callable pC.N(A );
    @*/
  public void M(C pC);
}

public class C {

  /*@
    @ protected code normal_behavior
    @    callable pA.M(C );
    @*/
  public void N(A pA);
}

public class B extends A {
  public void M(C pC) { ... pC.N(this); ... }
}

public class D extends C {
  public void N(A pA) { ... pA.M(this); ... }
}

public class Main {

  static int main (...) {
      C vC = new C();
      B vB = new B();
      vb.M(vC);  // calls B.M which calls C.N (code is not available)
  }
}
```

Figure 2.16: An example of a callback cycle involving methods of more than one class.

The subclass method `setFrom` super-calls `copyFrom` which immediately calls back down to `setFrom` thereby creating a nonterminating loop. To avoid such potential nontermination problems, subclass methods cannot make super-calls that insert a superclass method into a callback cycle. The following invalidation rule accomplishes this.

```
public class CellPlusPrevious extends IntCell {

  protected int _prevValue;

  ...
  public void setFrom(IntCell c) {
      _prevValue = _val;
      _val = c.getValue();
  }
  public int copyFrom(IntCell c) {
      setFrom(c);
      return getChange(); // expects any class invariants to hold
  }
}
```

Figure 2.17: A fragment of `CellPlusPrevious`'s implementation from the file `CellPlusPrevious.java`.

```
public class CellPlusInvariant extends CellPlusPrevious {

  protected int _diff;

  ...
  public void setFrom(IntCell c) {
      _diff = super.copyFrom(c);
  }
}
```

Figure 2.18: A fragment of `CellPlusInvariant`'s implementation from the file `CellPlusInvariant.java`.

Callback Cycle Invalidation Rule. Let *S* be a subclass of *C*. Let *P* be an overriding method in *S*. If a superclass method *M* directly or indirectly self-calls down to method *S*::*P*, then *C*::*M* cannot be super-called by *S*::*P*.

Notice that this rule does not invalidate method *M* for all subclass methods; it only says that a subclass method *S*::*P* cannot super-call methods that call back to *S*::*P*. This rule disallows the implementation in Figures 2.17 and 2.18 because superclass method `copyFrom` is invalidated with respect to the subclass method `setFrom`, i.e., the super-call in Figure 2.18 inserts a superclass method into a callback cycle which could, and in this case does, lead to nontermination.

2.6 Private Variables and Methods

In JML, variables and methods cannot appear in the public or protected specifications if they are not in scope in these specifications. For example, in Java, only fields and methods with public or protected visibility are visible to subclasses. Therefore, without some additional mechanism, private members would not be allowed to appear in the public or protected specifications used by our technique. However, private fields and methods need to be handled by our technique because, when superclass methods call methods or assign to variables that are not visible to subclasses, it is not possible to know the effect of these calls and changes on private instance variables without the superclass code or more information. The rest of this section provides examples that further illustrate the need for a way to handle the private fields and methods of a class and describes how our technique accomplishes this; we also describe an alternative way that private fields could be handled.

2.6.1 Maintaining private superclass fields

When creating a subclass, the customizer needs to know whether or not a superclass method is maintaining variables that are not visible to the subclass, e.g., assigns to private variables. A subclass needs to know when its superclass has such fields because the subclass may need to make super-calls in order to update those variables[28]. This information is needed when overriding a superclass method that has side-effects.

To illustrate the need for our special handling of fields that are not visible to subclasses, consider for example, the public and private specifications of class `CellPlusPrivate` given in Figures 2.19 and 2.20. Notice that this example is equivalent to class `CellPlusInvariant` given in Figures 2.4 and 2.5 except that `_diff` and the invariant are declared with private rather than protected visibility. Also, the protected specification would have to be empty because there are no protected fields or methods. If a new subclass of `CellPlusPrivate` needs to override method `setFrom`, then, because `oldVal` is assignable (see Figure 2.2), the concrete field `_diff` must also be changed. However, because `_diff` is not visible to the subclass, the overriding subclass method must make a super-call to `setFrom`, otherwise the superclass specification will not be satisfied, that is, the superclass invariant in Figure 2.20 will not be satisfied.

However, the customizer, who only sees the public and protected specification, will not know that `CellPlusPrivate` declares a private field (Figure 2.20 would not be available). This example shows that subclasses usually need to know about superclass variables that are not visible to the subclass because, in such cases, a subclass method must make super-calls whenever such superclass variables need to be maintained. Specifically, an overriding subclass method needs to make a super-call so the

28. Package visible fields would have to be handled, in our technique, in the same way as private fields because they are not visible to all subclasses. However, in this dissertation, we assume that classes do not declare package visible members (see assumptions in subsection 1.6.6 and explanation in subsection 2.9.5).

```
public class CellPlusPrivate extends CellPlusPrevious {

  /*@ public normal_behavior
    @   assignable value, oldVal;
    @   ensures value == initVal && oldVal == initVal;
    @*/
  public CellPlusPrivate(int initVal);
}
```

Figure 2.19: `CellPlusPrivate`'s public specification, from `CellPlusPrivate.jml-refined`.

```
//@ refines "CellPlusPrivate.java";

public class CellPlusPrivate extends CellPlusPrevious {

   private int _diff;
   //@         in  value, oldVal;

   //@ private invariant _diff == Math.abs(value - oldVal);
}
```

Figure 2.20: `CellPlusPrivate`'s private specification, from `CellPlusPrivate.refines-java`.

private instance variables satisfy the inherited specification, which includes the private specification (even though our technique does not use the private specification in reasoning because it is not seen by customizers).

Therefore, to handle such situations, our technique requires that all private variables be declared with the spec_protected modifier (see assumptions in subsection 1.6.6). This JML modifier allows private fields and methods to have private visibility in the Java implementation but protected visibility in JML specifications. Figure 2.21 shows what the protected specification would look like when the spec_protected modifier is used. Note that the invariant can now be included in the protected specification (as required by our assumptions) because `_diff` has protected visibility in this specification. Furthermore, the private specification of Figure 2.20 would now be redundant and would no longer be necessary. In this way, our technique always allows customizers to know about private fields so the customizer knows to make a super-call when these private fields need to be updated.

To ensure that the required super-calls are made, we now introduce the Mandatory Super-Call Rule.

```
//@ refines "CellPlusPrivate.jml-refined";

public class CellPlusPrivate extends CellPlusPrevious {

   /*@ spec_protected @*/ private int _diff;
   //@                                  in  value, oldVal;

   //@ protected invariant _diff == Math.abs(value - oldVal);
}
```

Figure 2.21: `CellPlusPrivate`'s protected specification, from `CellPlusPrivate.refines-java`.

Mandatory Super-Call Rule. Let *S* be a subclass of *C*. Let *V* be an instance variable declared in *C*. Let *V*'s data group contain private variables. If *S* has to implement a new or overriding method (or constructor) *M* that modifies *V*, then *S*::*M* can only modify *V* (along with the private variables in *V*'s data group) by directly or indirectly super-calling methods (or constructors) of *C*.

Our tool will enforce this rule by disallowing subclass methods from making direct assignments to a superclass field when it is a member of a data group *V* that also contains private superclass fields. This rule also means that when a class declares private fields, then more of its subclasses could become unimplementable; this is because the rule only requires a super-call when there are fields that cannot be directly assigned in the subclass. Furthermore, the needed superclass method may have been invalidated so the required super-call would not be allowed, and this may make the subclass specification unimplementable without superclass code (see subsection 2.9.3). This is also why we suggest that concrete fields be declared with protected visibility in extensible classes.

Note that the above rule also applies to indirectly declared variables of a private field; thus such indirectly declared fields have to be mapped into a data group so subclasses know when a super-call is required.

In summary, our technique requires that all private variables be declared with the spec_protected modifier so they can appear in the protected specification (see assumptions in subsection 1.6.6); this enables the customizer (and our tool) to determine when private (or package visible[29]) superclass fields have to be modified and maintained through super-calls.

2.6.1.1 An alternative approach

The use of the spec_protected modifier makes it possible to include the private implementation details of a superclass in the protected specification used by customizers. This assumption simplifies

29. For simplicity, we do not consider package (default) visible methods and variables in this dissertation (see assumptions in subsection 1.6.6 and the explanation in subsection 2.9.5).

and makes our soundness proof more regular. However, library providers may not want to make these details available to customizers. Therefore, this section presents an alternative way of handling private variables, i.e., allowing the above Mandatory Super-Call Rule to be applied but without exposing private variables to subclasses.

Our alternative approach is to place some restrictions on the way private fields are handled in the private specification[30]. First we would require that private (and package visible) fields be members of a public or protected data group visible to subclasses. We would also restrict the data group clauses allowed in the declaration of such fields so the customizer (and our tool) can determine whether a method assigns to them, i.e., to fields not shown in the public or protected specification. Furthermore, this has to be possible using the public and protected specification and without the code. Therefore, data groups listed in the maps and in clauses of a private concrete field declaration would have to satisfy three conditions[31]:

1. the data group must have public or protected visibility so it is visible to subclasses,
2. the data group must be declared in the same class as the fields it contains (not in a superclass), and
3. the data group must not contain any concrete fields visible to subclasses.

Therefore, the customizer can assume that a data group contains some unknown private fields if that group has no concrete member fields declared in the public or protected specification and there is at least one non-abstract method that can assign to members of that data group. However, when a type is an abstract class and all methods that can modify a data group are also abstract, then we assume the concrete fields in that data group have not yet been declared, i.e., the data group does not yet contain any concrete fields. If the type is an interface, then we always assume that the data group does not contain concrete fields (since all interface methods are abstract and concrete instance fields cannot be declared in an interface).

For example, the in clause of Figure 2.20 would not be allowed because `value` and `oldVal` were declared in a superclass (and they contain protected concrete fields from the superclass). Therefore, `_diff` would have to be a member of a newly declared public (or protected) data group.

Figures 2.22 and 2.23 show how the public and private specifications could instead be changed so `_diff` could be kept private. Using Figure 2.22 and the fact that there is no protected specification, the customizer can assume that `difference` contains concrete fields that are not visible to the subclass; that is, since the type is not abstract and `difference` has no additional public or protected members,

---

30. Private specifications are used in the verification of superclass methods. Ordinarily, they would not be made available to programmers creating subclasses; this is so private specifications can be altered without affecting the behavior of subclasses.

31. These conditions restrict the possible specifications allowed in JML and may require extra data groups. However, they do not restrict the kinds of data group relationships allowed for private fields.

```
public class CellPlusPrivate extends CellPlusPrevious {

  //@ public model int difference;
  //@                  in  value, oldVal;

  //@ public invariant difference == Math.abs(value - oldVal);

  /*@ public normal_behavior
    @   assignable value, oldVal;
    @   ensures value == initVal && oldVal == initVal;
    @*/
  public CellPlusPrivate(int initVal);
}
```

Figure 2.22: `CellPlusPrivate`'s public specification, from `CellPlusPrivate.jml-refined`.

```
//@ refines "CellPlusPrivate.java";

public class CellPlusPrivate extends CellPlusPrevious {

  //@ private represents difference <- _diff;

   private int _diff;
   //@         in  difference;
}
```

Figure 2.23: `CellPlusPrivate`'s private specification, from `CellPlusPrivate.refines-java`.

it must contain private variables (used to determine its abstract value). Furthermore, since `difference` is a member of both `value` and `oldVal`, these two data groups must also contain the same unknown variables. Therefore, whenever methods of a subclass of `CellPlusPrivate` need to modify `value`, `oldVal`, or `difference`, a super-call is required because these unknown variables (i.e. `_diff`) can only be modified by methods of `CellPlusPrivate`. In summary, the public invariant shown in Figure 2.22 can only be maintained via super-calls because `difference` must contain (depend on) private concrete fields.

2.6.1.2 Summary

The Mandatory Super-Call Rule requires a super-call when a subclass implements new or overriding methods that need to modify data groups containing private fields; this is so these variables, `_diff` in this case, will be properly maintained.

For example, suppose a subclass of `CellPlusPrivate` overrides `setFrom`. The Mandatory Super-Call Rule says that this overriding method must directly or indirectly super-call `setFrom` to properly update `_diff` because `value` and `oldVal` are assignable. However, we do not require that the super-call be to the method being overridden; we only require that the inherited public and protected specification, regarding the data group (or model field) that contains private fields, be satisfied through super-calls. Thus the inherited specification will automatically be satisfied since our assumption is that every superclass method satisfies its specification when that method has not been invalidated by the subclass.

The above rule has another implication for customizers. That is, when a superclass data group contains both protected and (possibly unknown) private concrete fields, there may be relationships between these fields as specified in the invariant and represents clauses. Our technique handles this like we did with side-effects on objects whose runtime type is unknown statically. For example, in subsection 2.4.3, we illustrated some of the problems that can occur when directly assigning to fields of objects other than the receiver. Recall that when the type of an object is a subclass of its static type, there could be unknown additional fields related to the superclass fields being modified (e.g., through a subclass invariant). Therefore, our technique requires that assignment to fields of an object, other than the receiver, be done via object-calls since these methods know how to maintain the unknown subclass variables. Our technique for handling unknown fields in the superclass is similar, i.e., assignment to fields in a data group containing private members must be done through super-calls since these methods know how to maintain these private data fields and the subclass cannot directly assign to them. The net effect for the customizer is that when a superclass data group contains both protected and private fields that are not visible to a subclass implementation, the protected fields become read-only, i.e., subclasses cannot directly assign to them. Our tool will enforce the Mandatory Super-Call Rule by not allowing the subclass to assign to these read-only fields[32].

2.6.2 Visibility of type invariants

We also believe that invariant relationships between variables should be visible to customizers. Therefore, our technique requires this (see assumptions in subsection 1.6.6), i.e., that all invariants have public or protected visibility (a private invariant clause would have to be redundant)[33]. However, in Figure 2.20, the invariant clause is private. Thus this invariant would have to be declared so it is

32. Thus our tool assumes that the private, package visible, and protected fields in a data group have an invariant relationship since determining this from the specification is not always easy or computable. However, the customizer may be able to determine by hand whether or not a protected field has to be considered read-only and can only be updated through super-calls. This is another reason why we believe superclass invariant and represents clauses should be visible to subclasses.

33. This visibility requirement for invariants is also important to our alias control technique described in Chapter 3.

visible to customizers; this was corrected in Figure 2.21 where this invariant is made visible to subclasses.

2.6.3 Private field accesses

For soundness, the accessible clause of the subclassing contract must be part of the public specification of public methods (see Chapter 4). However, private (or protected) variables cannot be shown in a public accessible clause because they are not in scope. However, these variables can be handled by our technique (as they are in the assignable clause). For example, a private variable *W* would have to be a member of the data group of a public variable *V*. Thus the accessible clause for a method *M* can list *V* whenever *M* reads variable *W*. Hence, when a model field is listed in the accessible clause, it denotes the concrete fields in the model field's data group.

Therefore, every private (or protected) field must be a member of a data group visible to subclasses if it is accessed by methods of the class. However, these fields already have to be members of a public data group so they can be initialized by a public constructor. For example, a private variable cannot be assigned to, even in a constructor, unless it is a member of a data group with at least the visibility of the method (or constructor) where the assignment occurs. Therefore, there will usually be a non-private data group that can be listed in the accessible clause in place of the private variable.

Note, however, that when *W* has private visibility rather than protected visibility, then our technique becomes more conservative, i.e., more constructors may be invalidated (see subsection 2.4.5) than would have otherwise been necessary, especially if *V*'s data group contains more variables than just *W*. For example, downcalls to methods that access *V* may invalidate constructors in cases where only those that access *W* need to be invalidated. This would be mitigated by our assumption that private fields be declared with the spec_protected modifier.

2.6.4 Private method calls

Private methods are those that can only be called by methods of the class where it is defined (i.e., methods with `private` visibility in Java[34]). In contrast, *non-private methods* can be called by methods of other classes (i.e., `public`, `protected`, and default (package) visible methods in Java).

Therefore, visibility control (in Java) prevents private methods from being called by subclass methods (or by methods of unrelated classes). Furthermore, in our technique, private methods are not in scope in the superclass specifications used by customizers; only public and protected specifications are available[35]. Thus our technique has to handle private method calls without listing them in the callable clause of the protected subclassing contract.

---

34. Our technique does not allow object-calls to methods with private visibility and side-effects since these methods cannot be overridden (see assumptions in subsection 1.6.6 and explanation in subsection 2.4.3). Therefore, such object-calls would not be permitted in C++ even though C++'s `friend` feature allows them. Using the above definition, these unoverrideable methods could therefore be considered "private" since our technique does not allow friend methods to (object-)call them.

We are able to do this because private methods are unoverrideable by subclasses[36]. That is, unoverrideable methods cannot be involved in downcall problems, unless they make downcalls. Therefore, in JML, private methods are handled by treating them as if they were inlined, i.e., each method call or variable access made by a private method *P* is treated (in the subclassing contract) as if it were made directly by the method that calls *P*. For example, suppose method *M* calls *P*. Method *P* will not appear in the specification used by our technique, but *P*'s method calls and variable accesses (those visible in the protected specification) will appear in *M*'s subclassing contract. Therefore, if *P* would have been invalidated by a new subclass, then *M* will automatically be invalidated; that is, those method calls or variable accesses that would have invalidated *P* will (as required for soundness) now invalidate *M*.

2.7 Concrete Data Refinement

So far we have assumed that the superclass's public and protected invariant is maintained by the new subclass. In this section, we explore the ramifications of changing the way data is represented.

Data refinement is a program transformation in which either one set of variables is replaced by a different set, or the set of variables is unchanged but their properties (e.g., invariants) are changed. Data refinement is usually used to make representations more concrete or more efficient [GM94b, GM94a, Mor94, MG90]. An example of data refinement, occurred in Figure 1.5 when `IntCell`'s model variable `value` was refined to the concrete variable `_val`.

In a data refinement, a relation between the old and new variable is specified; this relation can be used to show that no unexpected behaviors arise [GM94b, GM94a, Hoa72, Mor94, MG90]. In our example, this relation was specified by the in and represents clauses in the protected specification for `IntCell` (Figure 1.5).

We distinguish two kinds of data refinement: model and concrete. Figure 1.5 is an example of *model data refinement*, where model variables are refined to concrete variables. A model variable can be replaced by a concrete variable because model variables, by definition, need not be part of the implementation.

Concrete data refinement means refinement of concrete variables to concrete variables. In this section we explore concrete data refinement in a subclass, where its superclass's concrete instance variables are data refined by the subclass's concrete instance variables. In most OO programming languages, such as C++, Java, Eiffel, and Smalltalk, code inheritance does not allow an instance variable of the superclass to be replaced or removed, since this is not type safe in general. The problem

35. Although C++ allows friend methods to make object-calls to private methods, the object-calls that cause problems (those with side-effects) are disallowed by our assumptions given in subsection 1.6.6 (see also subsection 2.4.3).

36. Final methods, in Java, could be handled in the same way because they are also unoverrideable, but this is unnecessary because final methods with public or protected visibility can be listed in the code contract.

```
public class NewRepCell extends CellPlusPrevious {

  protected int _valDiff;
  //@               in oldVal;

  //@ protected represents oldVal <- _val + _valDiff;
}
```

Figure 2.24: Protected specification of `NewRepCell` from the file `NewRepCell.jml`. An example of concrete data refinement if, through method overrides, superclass variable `_prevValue` is ignored and no longer used.

```
public class RepChangeCell extends CellPlusPrevious {

  //@ protected represents oldVal <- _val + _prevValue;
}
```

Figure 2.25: Protected specification of `RepChangeCell` from the file `RepChangeCell.jml`. An example of concrete data refinement that changes the way `_prevValue` is used; such changes in the representation invariant are not allowed by JML.

is that inherited superclass methods would try to refer to missing or changed instance variables which would not be prevented by the type system.

However, the programmer of a subclass can choose to ignore a superclass variable and use a different data structure in the subclass as long as superclass methods are overridden such that unsafe variable accesses are prevented. Furthermore, such concrete data refinement may permit a subclass to be implemented when that subclass would otherwise be unimplementable without superclass code, i.e., because a super-call was required (Mandatory Super-Call Rule of subsection 2.6.1) to a method that had been invalidated by the subclass.

Figure 2.24 shows an example where the subclass `NewRepCell` uses a different representation of `oldVal`. Ignoring (i.e., replacing) the concrete variable of the superclass and using the subclass concrete variable is an example of concrete data refinement. In this case, the representation invariant of superclass `CellPlusPrevious` would no longer hold for subclass objects (since `_prevValue` is being ignored). Such a concrete data refinement would be useful if `NewRepCell` needs to extend the behavior of `CellPlusPrevious`, but the represents clause is missing from Figure 2.3 or the superclass representation is inefficient; then `NewRepCell` could ignore the superclass representation and use the one given in Figure 2.24.

Another kind of concrete data refinement occurs when the same superclass variables are used but the data's properties are changed, and thus the way the data is interpreted and manipulated is changed. For example, the subclass shown in Figure 2.25 interprets the value of `_prevValue` differently than the superclass. The superclass expects it to contain the previous value of the cell, whereas the subclass expects it to contain the difference between the previous value and the current value. Such a change of interpretation would also mean that the representation invariant of `CellPlusPrevious` would no longer hold for subclass objects.

As mentioned earlier, situations that require a concrete data refinement could occur accidentally when the documentation for the superclass is incomplete. However, concrete data refinement can only be safe if certain restrictions are followed and the necessary methods are overridden. For example, in both of the previous examples, all methods that access `_prevValue` must be overridden because the superclass representation invariant involving that variable no longer holds.

Our purpose in exploring concrete data refinement is to explore the ramifications of:

1. not providing the representation invariant to programmers reusing a software framework or class library (i.e., keeping the representation private),
2. making data structure changes to improve efficiency in comparison to inherited data structures, and
3. not being able to update private superclass fields because the required super-calls are unsafe, i.e., the superclass methods have been invalidated by our rules.

The next rule must be considered when a subclass method directly accesses or modifies concrete variables of the superclass and either there is no protected invariant for the superclass or the subclass's protected invariant does not imply the superclass's protected invariant. Recall that we assume that the class invariant is visible to all subclasses, that is, there are no private invariants in the superclass (see assumptions in subsection 1.6.6). “Direct access” means access that names specific instance variables; these may be instance variables of the receiver or instance variables of another object. An access of a field of the receiver object will be called a *self-access*; an access of a field of an object other than the receiver is an *object-access*.

Data Refinement Overriding Rule. Let *S* be a subclass of *C*. A method, *M*, must be overridden if (i) *C*::*M* makes a direct self-access or object-access to a concrete variable *V* that is data refined by *S* and (ii) the part of *C*'s invariant[37] that concerns *V* is not maintained by *S*.

The above rule is concerned with accesses to variables in objects that are possible subclass objects; these accesses are exactly those included in the accessible clause (see Chapter 4). Other kinds of variable accesses are not allowed (see assumptions in subsection 1.6.6).

37. The represents clause specifies an invariant relationship between a model field and concrete fields; therefore, we consider the represents clause to be a part of the superclass invariant that needs to be maintained unless the subclass is doing a concrete data refinement.

The concrete data refinement rule mandates that superclass methods be overridden when they are expecting a different representation. For example, it would be unsafe to change the way the concrete variable `_prevValue` is interpreted and used in a subclass without overriding all superclass methods that access that variable. Furthermore, calling such superclass methods must not be allowed. The following rule invalidates these methods.

Data Refinement Invalidation Rule. A superclass method must not be super-called if it had to be overridden based on the Data Refinement Overriding Rule.

Although concrete data refinement requires that unrelated classes do not access instance variables of a given class, several OO programming languages allow such access (see also assumptions in subsection 1.6.6). For example, Java allows such access within packages, and C++ allows it with its friend feature. Our rule for concrete data refinement can only ensure that methods of related classes do not access concrete variables whose interpretation has changed because of a concrete data refinement; they do not prevent unrelated classes from accessing these variables. Therefore, JML only allows such concrete data refinements if the invariant of the subclass implies the invariant of the superclass. Indeed, JML's use of specification inheritance forces the subclass to maintain the superclass's invariant and representation.

However, we did not limit our study to JML. For example, Smalltalk's scope rules only allow methods to access variables of the receiver object; thus Smalltalk prevents methods from accessing variables of objects of unrelated classes. Also since Smalltalk methods are all public, these programs do not require any special (manual or automatic) static analysis to ensure that a concrete data refinement can safely be done through our technique of method overrides.

2.8 Super-Calls

Often it is convenient to call a superclass method when making a minor extension to a method in a subclass. Furthermore, in some cases it can be mandatory that the superclass method be called. For example, an overriding method must make super-calls if the superclass representation is private or hidden from the subclass (see Section 2.6 above).

As illustrated by the examples given in this chapter, super-calls are not always safe. The solution to this, and to other similar downcall problems, is to prevent super-calls to methods that have been invalidated by the new subclass. The invalidation rules have been given for this purpose; they specify when a superclass method might no longer satisfy its superclass specification or, because of downcalls, it might have unverifiable (without superclass code) additional side-effects.

Some languages with multiple inheritance, such as C++, permit calls to methods of any named superclass; such calls are also super-calls and need to take the above rules into account. When applying invalidation rules, one needs to use the subclassing contract of the superclass method being called; this is because a method *B*::*M* could be invalidated while the method it overrides, *A*::*M*, is not invalidated by a new subclass (e.g., if *A*::*M* does not make any downcalls but *B*::*M* does).

The authorization rule below is based on the following reasoning. Suppose a superclass method *M* has been invalidated by the new subclass and thus should not be executed. If *M* has been overridden by the subclass, then the only way it can be invoked is via a super-call (on an object of the subclass). Therefore, super-calls to *M* must also be prevented. The following rule ensures that all super-called methods satisfy their specifications and have no additional side-effects.

Super-Call Authorization Rule. Let *S* be a subclass of *C*. A superclass method *C*::*M* may only be called by subclass method *S*::*P*, if *C*::*M* has not been invalidated for *S*::*P*.

Most invalidation rules invalidate a superclass method for all subclass methods. However, the Callback Cycle Invalidation Rule invalidates a specific superclass method with respect to a specific subclass method. Therefore, the above rule must also do the same, e.g., it disallows specific super-calls (*C*::*M*) from within a specific subclass method (*S*::*P*).

2.9 Discussion

In this section we discuss some consequences of our rules. The first subsection discusses the consequences of an overriding method that does not satisfy the specification of the method it overrides; although JML does not allow this, we provide a set of rules to handle such situations and describe the motivation for and usefulness of those rules in OO languages like C++. The second subsection discusses the consequences of and problems related to unoverrideable methods. The third subsection then describes those situations where a correct subclass cannot be written without superclass code. The fourth subsection revisits our invalidation rules and shows why they can sometimes be more conservative than necessary. The fifth subsection discusses the subclassing contract as a specification, and finally, the last subsection summarizes the results of our study of downcalls.

2.9.1 Non-Refining Methods

This section describes an overriding rule that deals with non-refining subclass methods, that is, methods that do not conform to the superclass's specification. The notion of refinement relates behavioral specifications as defined in Figure 1.1. A subclass method that refines the method it overrides will be called a *refining method*, otherwise, an overriding method is a *non-refining method*. A new subclass method that does not override a superclass method is neither a refining nor a non-refining method.

Method refinement, in contrast to behavioral subtyping [Ame91, DL96, LW93, LW94], is defined from the point of view of the implementer rather than that of the client. Thus the protected specification, rather than the public specification, must be used in reasoning about method refinement. The protected specification combines the specifications that have either public or protected visibility. For example, the protected specification for `IntCell` includes everything in Figures 1.2, 1.5, and 1.10. Notice also that the subclassing contract specifies properties of a method’s behavior, so it must

also be included in the reasoning about method refinement (see Chapter 4 for more details about the use of the subclassing contract).

A superclass method may not behave as expected if it calls down to a non-refining method. For example, suppose *S* is a subclass of *C*. If *S*::*P* is a non-refining method, then *S*::*P*'s allowed behavior is not a subset of the allowed behavior of *C*::*P*, when *C*::*P*'s precondition holds. Therefore, superclass methods that call down to *S*::*P* may not behave as specified, since they were verified using the superclass specification of *C*::*P*. For example, Figure 2.26 gives an example of a subclass of `IntCell` with a non-refining method. Method `setValue` is non-refining because it does not change `value` when the parameter `newVal` is less than zero, contrary to its superclass specification. The implementation of `IntCell` shown in Figure 2.27 illustrates the problem; that is, method `setFrom` will no longer satisfy its specification because of its downcall to `setValue`, which also does not behave as expected (i.e., `setValue` does not satisfy the superclass specification).

A downcall to a non-refining method can happen either as a self-call or via a subclass object-call. A *subclass object-call* is an object-call in which the dynamic type of the receiver could be a subtype of the current class. For example, the object-call `o.setValue(v)` calls a non-refining subclass method when `o` has type `NonRefiningCell`. This expression will not satisfy the superclass specification. Therefore, any method containing a self-call or a subclass object-call to a non-refining method (like `setValue`) may not behave as expected and must be overridden. The following rule ensures that this is done.

Non-Refining Method Overriding Rule. A superclass method must be overridden if it makes a direct self-call or a subclass object-call to a method that has been overridden by a non-refining method.

This rule prevents unexpected and incorrect behavior for a particular subclass. However, in general it is unsound to allow non-refining public methods in subtypes, since this breaks behavioral subtyping. Therefore, if a subclass has a non-refining method, then subclass object-calls to that method must be prevented everywhere in the system; our rules and assumptions prevent them within the subclass, but not elsewhere in the system. For example, methods must not be allowed to pass such subclass objects directly or indirectly (inside a data structure such as an array or object) to a method that expects an object that satisfies the superclass's specification. This is why JML enforces method refinement (and behavioral subtyping) through specification inheritance [DL96].

The above rule requires an override because a superclass method that calls down to a non-refining method may no longer behave correctly with respect to its specification. However, in addition, such methods must not be super-called by subclass methods. The following rule invalidates such methods and disallows implementations that make such super-calls.

Non-Refining Method Invalidation Rule. A superclass method must not be super-called if it makes a direct self-call or a subclass object-call to a method that has been overridden by a non-refining method.

```
public class NonRefiningCell extends IntCell {

  ...

  /*@  public normal_behavior
    @     requires newVal >= 0;
    @     assignable value;
    @     ensures value == newVal
    @    also
    @     requires newVal < 0;
    @     ensures \not_modified(newVal);  @*/
   public void setValue(int newVal);
}
```

Figure 2.26: Part of the public specification of `NonRefiningCell`.

```
public class IntCell {

  protected int _val;

  public void setValue(int newVal) {
      _val = newVal;
  }
  public void setFrom(IntCell c) {
      setValue(c.getValue());
  }
  ...
}
```

Figure 2.27: A fragment of `IntCell`'s implementation from the file `IntCell.java`.

JML avoids the problems illustrated in this section by enforcing method refinement through specification inheritance. However, in languages like C++ one can create a subclass that is not a subtype by using protected or private inheritance. Since the new subclass is not a subtype, the type system does not allow these incorrect subclass object-calls. Therefore, in such cases, one will sometimes override a method in a way that makes it non-refining. Thus, these rules are also useful in such languages because they prevent unexpected behavior within these (non-subtype) subclasses.

Our tool, however, will not enforce these rules because they require reasoning about and comparing the superclass and subclass method specifications to determine whether or not a method is a

refining method. That is, refining methods are only allowed to strengthen the specifications of the methods being overridden, which would require a proof. Furthermore, JML requires and our tool assumes that all overriding methods are refining methods.

2.9.2 Unoverrideable Methods

From the point of view of a new subclass, an *unoverrideable method* is a method that it cannot override. For example, final methods of a superclass may not be overridden in Java, and non-virtual methods may not be overridden in C++. Methods of a superclass may also be unoverrideable because of visibility control; for example, private methods are unoverrideable in Java and C++; static methods and constructors are also unoverrideable. Further, from the point of view of a new subclass, S, methods in classes that are unrelated to S are also unoverrideable. The following rule deals with invalidated, non-public methods that are unoverrideable; its purpose is to prevent such methods from being invoked.

Unoverrideable Method Rule. Let M be a non-public superclass method that is not (object-) called by methods of an unrelated class. If M cannot be overridden and is invalidated by the new subclass, then all methods that directly call M must be overridden and M cannot be super-called by subclass methods.

Notice that the above rule is both an overriding rule and an invalidation rule, that is, it mandates a method override of and invalidates method M. Furthermore, it only applies to non-public methods. If the rule allowed M to be public, then behavioral subtyping would fail for the subclass. This is because when a subclass inherits an unoverrideable, public method, there is no way to prevent clients from calling it, and when the inherited method has been invalidated there is no way to guarantee that such a method meets its specification (without superclass code). However, in a language like C++, one can use protected inheritance without worrying about behavioral subtyping (since such subclasses are not subtypes and superclass methods are not visible to clients unless the subclass allows it); in such cases, this rule is useful in preventing invalidated, inherited methods from being called.

In addition, a protected method M, in Java, can be (object-) called by unrelated classes within the same package. However, if M is called by a method of an unrelated class, then a subclass S must be considered unimplementable if M is unoverrideable and has been invalidated by S. This is why our technique does not allow this situation; that is, our assumptions (see subsection 1.6.6) do not allow object-calls to methods that might be invalidated by a new subclass, i.e., our assumptions do not allow object-calls that introduce mutual recursion between methods of unrelated classes or object-calls to unoverrideable methods with side-effects[38].

38. Our assumptions do not disallow all callbacks between objects of unrelated classes, only those that introduce mutual recursion. Further, requiring that methods with side-effects be overrideable is not overly restrictive since this is the case for all methods in Smalltalk.

2.9.3 Unimplementable Subclasses

An interesting result of our study is that in some situations it is not possible to write a verifiably correct implementation of a subclass specification without changing or at least seeing the superclass's code. One such situation occurs when a public method is invalidated but that method is unoverrideable. For example, if method `setFrom` of `IntCell` were final, then the subclass `CellPlusPrevious` would not be implementable without changing the superclass code (e.g., removing the final modifier). This is why it is our belief that methods with side-effects should always be overrideable.

Subclasses can also be unimplementable in situations like the following. Suppose some part of the superclass object state is private and a subclass method needs to update private superclass variables to satisfy the subclass specification. If there is no superclass method that modifies the required variables or if all such methods have been invalidated by the new subclass, then a provably correct implementation satisfying the subclass specification may not be possible without superclass code (although there may be some cases where the reasoning technique given in Section 2.8 may allow it). Consider for example, Figure 2.28. The subclass `ReversibleCell` would be unimplementable if superclass variable `_totalChg` had private rather than protected visibility. Specifically, method `reverse` would not be implementable since there is no superclass method that sets `_totalChg` to a specific value.

A new subclass *S* can also be unimplementable when a method *M* of an unrelated class is invalidated by *S*. For example, *S* would be unimplementable if it adds a new subclass invariant when *M* makes direct assignments to a field of an object that has static type *T* but could have dynamic type *S* (see the example in subsection 2.4.3). Such a subclass *S* would be unimplementable without the code for *M* or the code for superclass *T*. A similar situation also occurs when the code for *M* is unavailable and *M* is mutually recursive with overridden methods in *S* (see also the examples in Section 2.5).

One way to avoid most of these problems is to make all methods of an extensible class overrideable, and to allow subclasses to directly access all superclass instance variables. For example, in Java, methods are overrideable unless they are declared to be final; in C++, methods are overrideable when they are declared as virtual. Also, subclasses can access instance variables when they are declared with protected rather than private visibility in Java or C++. Furthermore, in Smalltalk, all methods can be overridden and superclass instance variables of the current receiver are visible to subclass methods.

However, if an extensible superclass does have private variables, then methods that assign to these variables should not make calls to non-private methods; this restriction prevents such superclass methods from being invalidated. For example, superclass methods can be invalidated by calls to non-private methods because non-private methods can be overridden which would introduce downcalls;

```
public class ReversibleCell extends CellPlusTotal {

  /*@ public model int oldValue;
    @                       in value;
    @*/
  /*@ public model int oldTotal;
    @                       in totalChg;
    @*/

  /*@  public normal_behavior
    @     assignable value, totalChg, oldValue, oldTotal;
    @     ensures value == initVal && oldValue == initVal
    @          && totalChg == 0 && oldTotal == 0;
    @*/
  public ReversibleCell(int initVal);

  /*@ also
    @  public normal_behavior
    @     assignable oldValue, oldTotal;
    @     ensures oldValue == \old(value) && oldTotal == \old(totalChg);
    @*/
  public void setValue(int newVal);

  /*@ also
    @  public normal_behavior
    @     requires c != null;
    @     assignable oldValue, oldTotal;
    @     ensures oldValue == \old(value) && oldTotal == \old(totalChg);
    @*/
  public void setFrom(IntCell c);

  /*@  public normal_behavior
    @      assignable value, totalChg;
    @      ensures value == \old(oldValue) && totalChg == \old(oldTotal)
    @           && oldValue == \old(value) && oldTotal == \old(totalChg);
    @*/
  public void reverse();
}
```

Figure 2.28: `ReversibleCell`'s public specification, from `ReversibleCell.jml-refined`.

these downcalls may require that a new subclass invariant hold and they could introduce additional side-effects.

2.9.4 Invalidation Rules Revisited

In this subsection, we show that, for some specific cases, our invalidation rules are more conservative than necessary. In particular, we give examples that show that the correctness of a subclass method can sometimes be proven even though the subclass makes a super-call that is not allowed by one of our rules. We first revisit the Additional Side-Effects Invalidation Rule, followed by the constructor invalidation rules.

2.9.4.1 The Additional Side-Effects Invalidation Rule revisited

In this subsection, we analyze the Additional Side-Effects Invalidation Rule, given in subsection 2.2.3, to determine when a subclass is not required to follow this rule, i.e., when a correctness proof is possible even though the rule has been violated. We also explain why we use our rule rather than a more liberal one.

The Additional Side-Effects Invalidation Rule says that a superclass method *C*::*M* must not be super-called if it makes a downcall that may have additional side-effects on field *W*. However, *C*::*M* might still satisfy its superclass specification since its effects on *W* are not and cannot be specified in the superclass. Therefore, from a practical standpoint, one may still be able to reason about a super-call to *C*::*M* using the superclass specification. Nonetheless, without the superclass code (or more information), when *C*::*M* has been invalidated by this rule, it is not generally possible to know what effect calling *C*::*M* has on *W*.

For example, Figure 2.29 shows a different implementation of `setFrom` than the one originally given in Figure 1.7 (the superclass implementation is given in Figure 1.8). This new implementation is correct even though the super-call to `setFrom` would not be allowed by the Additional Side-Effects Invalidation Rule of subsection 2.2.3. The superclass implementation of `setFrom` satisfies its specification because the downcall to method `setValue` satisfies the superclass specification for all values of `_totalChg`, the variable that may have been changed during the downcall. However, `_totalChg` cannot be used after the super-call because, without the superclass code, one does not know under which conditions or how many times the superclass method `setFrom` will make a downcall to `setValue` (and hence, what the value of `_totalChg` will be after the super-call). Furthermore, there is no subclass invariant involving `_totalChg` so changes to that variable do not affect the precondition of any of the downcalls made. Therefore, the subclass implementation of `setFrom` (Figure 2.29) can satisfy its postcondition by not directly or indirectly using `_totalChg` after the super-call to `setFrom`.

Therefore, one can reason about super-calls in subclass methods as follows. Suppose *C*::*M* has additional side-effects on *W* and makes downcalls. If all subclass methods downcalled by *C*::*M* satisfy the superclass specification for all possible values of *W*, then *C*::*M* will also satisfy its specification (since *C*::*M* was verified using the superclass specification). Notice also that all values of *W* have to satisfy the subclass invariant, otherwise the preconditions of the downcalled methods may not be

```
public class CellPlusTotal extends IntCell {

  protected int _totalChg;

  // ...

  public void setValue(int newVal) {
      _totalChg += Math.abs(newVal - _val);
      super.setValue(newVal)
  }
  public void setFrom(IntCell c) {
      int newTotal = _totalChg + Math.abs(c.getValue() - _val);
      super.setFrom(c);
      _totalChg = newTotal;
  }
}
```

Figure 2.29: A correct implementation of `CellPlusTotal`, based on the additional side-effects authorization rule, from the file `CellPlusTotal.java`.

```
public class CellPlusPrevious extends IntCell {

  protected int _prevValue;

  // ...

  public void setValue(int newVal) {
      _prevValue = _val;
      super.setValue(newVal)
  }
  public void setFrom(IntCell c) {
      int previous = _val;
      super.setFrom(c);
      _prevValue = previous;
  }
}
```

Figure 2.30: A correct implementation of class `CellPlusPrevious`, based on the additional side-effects authorization rule, from file `CellPlusPrevious.java`.

satisfied. Therefore, if *C*::*M* has not been invalidated by any of the other rules, it may still satisfy its superclass specification and its behavior can be reasoned about using this specification. However, after

making a super-call to *C*::*M*, a subclass method must not directly or indirectly use the value of any subclass variable (such as *W*) that may have been modified by *C*::*M* (since its specification does not mention subclass variables).

Another example where this reasoning technique can be applied is given in Figure 2.30; it shows how the subclass method `setFrom` of `CellPlusPrevious` can be correctly implemented using a super-call when the above restrictions are followed. That is, `setValue` of the subclass satisfies its superclass specification for all possible values of `_prevValue`, and the value of `_prevValue` (and `oldVal`) is not used in `setFrom` after the super-call; the value of `_prevValue` must be treated as if it were unknown since it might have been changed during the super-call.

To summarize, one can reason, using the superclass specification, about the behavior of a method *C*::*M* that makes downcalls that can have additional side-effects on *W* if:

1. all methods downcalled by *C*::*M* satisfy their superclass specifications for all values of *W*,
2. the subclass invariant holds for all values of *W*, and
3. the value of *W* is not used after a super-call of *C*::*M* or until *W* has been updated in the subclass.

The reason the Additional Side-Effects Invalidation Rule is not based on the above reasoning is because violation of such a rule would be difficult to detect statically, that is, it requires a proof that downcalled methods satisfy the superclass specification. However, the Additional Side-Effects Invalidation Rule of subsection 2.2.3 is easy to enforce using the subclassing contract; therefore, our tool will enforce this more restrictive rule.

2.9.4.2 The constructor invalidation rules revisited

Customizers can also work around and sometimes avoid the problems related to the constructor invalidation rules given in subsection 2.4.5. For example, if a superclass constructor makes downcalls to methods that reference uninitialized subclass fields, the customizer can create and call a private subclass helper method as one of the constructor's actual parameters, e.g., `super(helper())`; this helper method would be invoked before the superclass constructor call, and would initialize those subclass fields and establish any required subclass invariants.

Nonetheless, without superclass code, the invalidation rules for methods would still apply to superclass constructors, i.e., the above coding trick only initializes subclass fields and establishes the subclass invariant which does not prevent the other problems related to super-calls. The invalidation rules for constructors have to be more restrictive than the rules for methods because the class invariant is not established prior to constructor calls.

Furthermore, if the superclass invariant implies the subclass invariant, then the Invariant Invalidation Rule (subsection 2.4.1) and the constructor Invariant Invalidation Rule (subsection 2.4.5) would no longer apply since the subclass invariant would automatically be established by the

superclass invariant. Thus our rules assume that subclass invariant clauses are not redundant and specify additional constraints on fields.

Also, the above coding trick points out another potential problem with superclass constructor calls, i.e., a method super-call could be made during the evaluation of an actual parameter being passed to a superclass constructor, e.g., `super(super.m());` thus such super-calls are unsafe since they would occur before the superclass has been properly initialized. However, this should be rare in practice, so we make the assumption that no super-calls are not made prior to execution of a superclass constructor (see assumptions in subsection 1.6.6).

2.9.4.3 Summary

These examples illustrate that our rules are conservative. Therefore, it may be possible for customizers to find other ways of reasoning about superclass methods that have been invalidated by one of our rules. That is, if the customizer can prove correctness without the superclass code, then the rules may not have to be followed in all cases. Nonetheless, our rules are such that, if followed, the customizer will be guaranteed that a correctness proof of the subclass can be obtained without superclass code. Also, our tool will warn customizers when there are potential problems so they can be avoided when possible.

2.9.5 Package Visible Fields and Methods

For simplicity, we do not consider package visible instance fields and methods in this dissertation. However, these members could be handled by our technique. That is, they could be handled like a special category of private and protected members. They have to be handled like private members because they are not visible to all subclasses, i.e., when the superclass and subclass are in different packages. Furthermore, package visible members also have to be treated like protected members, in some cases, because they are visible to subclasses when the superclass and subclass are in the same package. Also, if a superclass has package visibility, then all of its subclasses have to be in the same package; thus all package visible members of a package visible superclass have to be handled like protected members by its subclasses.

For example, package visible members cannot appear in the protected subclassing contract of subclasses when the superclass and subclass are in different packages; however, subclasses need to know when they exist. That is, all subclasses need to know about package visible fields so subclasses know when a method can safely be super-called and when a super-call is required (see subsection 2.6.1). Therefore, package visible concrete fields would have to be handled like private concrete fields when they are not visible to the subclass (i.e., when the superclass and subclass are in different packages). That is, like our requirement for private concrete fields, package visible fields would have to be declared with the spec_protected modifier so they are visible to all subclasses.

Similarly, package visible methods would have to be treated as if they were inlined in the protected callable clause when the superclass and subclass are in different packages, since these

methods would not be visible or overrideable in those subclasses. However, subclasses in the same package would need a package visible subclassing contract where package visible members could be listed in the callable and accessible clauses; this is necessary for reasoning about overriding and invalidation of package visible superclass methods. Therefore, package visible methods would have to be handled like protected members when the subclass is in the same package (since the subclass can override them), but like private members when the subclass is in a different package.

To avoid the added complexity of having two different specifications when a class declares package visible methods and fields, we assume, in this dissertation, that classes do not declare package visible members (see assumptions in subsection 1.6.6). Furthermore, in extensible classes, we believe that all such fields and methods should be declared with protected visibility so they are visible to all subclasses (see Chapter 4); thus if the designer does not want a method to be visible to subclasses in a different package, then it should probably be private. We leave additional details as future work.

2.9.6 The Subclassing Contract as a Specification

Specifications are meant to be abstract and can usually be implemented in many different ways. In our study of the problem caused by downcalls, we found that more information was needed than is provided in a "standard" specification. However, our goal was not to overspecify or to be overly restrictive in the implementations permitted by this additional information. Therefore, in this section, we first consider the properties of specifications and then whether the clauses in the subclassing contract are abstract enough to be considered a specification, and whether they disallow too many possible implementations.

2.9.6.1 Properties of specifications

Programming is a process that starts with a specification and ends with an implementation, i.e., a program that satisfies the specification. A specification describes the requirements of a system in terms of its behavioral attributes and is more abstract than the implementation. The implementation defines a sequence of steps, an algorithm, that can be executed on a computer system to accomplish a specific result or task. However, the specification "abstracts out" the details that are unnecessary for describing and understanding the behavioral attributes required by users and clients of the system. For example, the implementation may use some arcane, but efficient, data structure and algorithm; however, the specification and user need not be concerned with such details.

The specification describes "what" the implementation is supposed to do, whereas the implementation describes "how" that specified behavior is to be accomplished. So a specification is nonprocedural in contrast to the procedural, algorithmic, executable nature of the implementation.

Nonetheless, the specification is closely related to the implementation in that the implementation must satisfy the specification. However, a specification, because it is more abstract, can be satisfied by many different implementations.

A specification may include not only the functional behavior of the system, but also other properties. For example, a specification may include performance requirements such as the minimum response time to certain events, or the minimum number of transactions that need to be processed during a given time frame; it could also specify reliability requirements such as the maximum allowable data loss due to system failures (either hardware or software). A specification may also include security requirements such as who is allowed and how one gains access to the system.

A specification also serves as a contract between the user and implementer; it specifies the obligations required of both the user and implementer, and the results that both sides can expect. For example, the precondition of a method specifies the obligations of the user, i.e., what the caller must guarantee before making the call; the precondition also specifies what the implementer can expect when the call is made. On the other hand, the postcondition specifies the obligation of the implementer, i.e., what the implementer guarantees when the precondition is satisfied; the postcondition also specifies what the user can expect after the call. In JML, the assignable clause specifies the only variables that can be changed during the execution of a method; it specifies the obligation of the implementer and what the user can expect.

In summary, a specification has the following characteristics. A specification

1. is abstract and nonprocedural,
2. allows multiple implementations,
3. can specify properties other than functional behavior,
4. serves as a contract between the user and the implementer.

2.9.6.2 The subclassing contract

The subclassing contract is for use by customizers extending the behavior of classes in a reusable framework or class library; it specifies the methods that may be called and the fields that may be accessed. Customizers use this information to determine when there may be downcalls that could cause the subclass to behave incorrectly, for example, when a superclass method may no longer satisfy its specification; in such situations, the superclass method cannot be super-called and must be overridden by the subclass.

Another purpose of the subclassing contract is to impose these restrictions on method calls and field accesses when a superclass method is reimplemented or modified. That is, when the code of a superclass method is changed, even if it satisfies its behavioral specification, this method may still cause subclasses to behave incorrectly unless it also satisfies its subclassing contract. For example, if it calls methods not listed in the callable clause, this may result in downcalls that were not considered when a subclass was originally implemented. Therefore, the subclassing contract must guarantee that these methods and fields are the only ones called and accessed so new downcall problems are not created by changes in the superclass implementation.

Another way to view this would be that if a superclass method is reimplemented or modified and the new implementation does not satisfy its current subclassing contract, then, using our rules, all subclasses would have to be re-evaluated using the new subclassing contract. If this new subclassing contract includes any new calls or field accesses, then additional superclass methods may need to be overridden and some super-calls may no longer be allowed. Furthermore, the framework or class library providers would have to notify all their clients that a change was made in the library implementation that could affect the correctness of subclasses (or they could change the implementation so it does not violate its current subclassing contract).

However, it is also important to note that the subclassing contracts of the superclass also limit the callable methods or accessible fields of an overriding subclass method. That is, an overriding subclass method is not permitted to make calls that are not permitted by the method it overrides. For convenience, the subclassing contract can be derived from the code of the superclass method, but an overriding subclass method must also satisfy this generated specification[39].

To summarize, the subclassing contract is used by customizers when creating subclasses. Our rules together with the subclassing contract alert customizers when there may be problems due to downcalls whether the superclass code is available or not; they can also be used to alert framework providers when changes in the implementation of library classes may cause subclasses to behave incorrectly so they can notify clients and be specific about where the potential problems may occur. For these reasons, we believe that the subclassing contract serves as an important part of the specification of extensible classes in a reusable framework or class library.

Finally, the subclassing contract has the four characteristics of a specification as described in the previous subsection. It is abstract and nonprocedural in that it specifies the methods that may be called and the variables that may be accessed; it does not specify how many times or when a method or variable is referenced. Therefore, many different implementations will satisfy a given subclassing contract. In fact, a new implementation is not required to make the method calls or variable accesses specified in the subclassing contract. The subclassing contract specifies the behavioral properties, in addition to the functional behavior, needed by customizers particularly when the superclass code is unavailable. The subclassing contract also guarantees that the methods and fields listed in the callable and accessible clauses are the only ones called or accessed by the specified method; it is a contract between the implementer and customizer of library superclasses.

2.9.7 Summary

If superclass code is unavailable when creating subclasses, our experience has demonstrated that (public and protected) behavioral specifications are not sufficient to avoid downcall problems. In the previous sections, we have provided examples to illustrate when and why downcalls can lead to

39. Chapter 4 provides more details about how best to generate and specify a subclassing contract that would not invalidate more methods than necessary.

unexpected behavior or nontermination. We have also shown that the calling structure, as specified in the callable clause of the subclassing contract, is needed so programmers know when a method override introduces a downcall. With this information, our rules can then be used to prevent downcalls from causing problems.

Unexpected behavior and nontermination can occur when superclass and subclass methods interact in certain ways. A superclass method interacts with subclass methods through downcalls; our examples show that problems can arise when an overriding subclass method modifies and, in some cases, accesses subclass instance variables. Thus the first downcall problem we investigated involved data group dependencies between superclass and subclass variables. The in and maps clauses declare a data group relationship and allow overriding subclass methods to modify subclass instance variables. Our technique avoids most of these problems through the application of the additional side-effects overriding and invalidation rules. However, some problems can only be avoided through a programming discipline as given in the assumptions of subsection 1.6.6, e.g., only fields of the current receiver object may be assigned to. Also, the JML semantics permits our technique to detect some potential problems because it does not allow temporary side-effects without listing those variables in the assignable clause.

The second problem we investigated involved possible nontermination due to mutual recursion between superclass and subclass methods, i.e., involving a callback cycle. To avoid such callback cycles, we introduced an overriding and an invalidation rule to ensure that any mutual recursion involves only superclass methods or only subclass methods. We also had to introduce an assumption to prevent callback cycles between methods of unrelated classes.

Finally, for completeness, we investigated method refinement and (concrete) data refinement. JML and our technique do not allow non-refining subclass methods. However since C++ allows protected and private inheritance, we gave rules for handling inherited methods that make downcalls to non-refining methods; these methods are not visible to clients of a new subclass and must not be called by those methods that are visible to clients. We also gave rules that could be used to allow concrete data refinement when the superclass representation is hidden, left unspecified, or contains private fields that cannot be modified through super-calls (they have been invalidated).

In Chapter 5 we present some guidelines and general design advice for implementers of frameworks and class libraries. These guidelines build on the ideas given in this chapter and Chapter 3, and they simplify reuse of extensible classes; they also help prevent subclasses from being unimplementable. Furthermore, when these guidelines are followed, most of the rules presented in this chapter become unnecessary. This greatly simplifies the reasoning required by customizers of frameworks and class libraries.

We also claim that if our rules can be followed, then the new subclass is implementable and verifiable without superclass code. For example, an overriding rule cannot be followed if a method override is required but that method is unoverrideable. Similarly, an invalidation rule cannot be

followed if the implementation of the subclass requires a super-call but that superclass method has been invalidated. In Chapter 4, we prove the soundness of our technique and that this claim is true.

However, conversely, if our rules cannot be followed, then the new subclass is usually (but not always) unimplementable and unverifiable without superclass code (or more information than is provided in the subclassing contract). Nonetheless, since our rules are conservative, not following them does not necessarily mean that the subclass will not behave correctly, but only that this cannot, in general, be verified without the superclass code (the counter examples given together with the rules show why the superclass code would be needed).

2.9.7.1 Comparison with our previously published work

This chapter refines and gives a more detailed explanation of the overriding rules we presented in a previous paper [RL00]. Also, in the previous paper, a superclass method was invalidated if it had to be overridden based on one of the overriding rules. In this chapter, we instead give an invalidation rule corresponding to each of the overriding rules; this allowed us to more precisely define when a superclass method should not be super-called, and thus, in some cases, our criteria for invalidating a superclass method became slightly less conservative. We have also combined the additional side-effects rule with what was previously called the invariant rule, since the problems, in both cases, are related to and caused by additional side-effects. The other overriding rules have not been changed much, if at all, although the explanation of them has been expanded. However, the explanation of how we prevent the invalidation of subclass invariants in objects other than the receiver, e.g., explicit parameter objects, is mostly new in this chapter (subsection 2.4.3). Recall that we prevent invalidation of subclass invariants by requiring that assignment to fields of objects, other than the receiver, be done through object-calls to non-private (overrideable) methods. We have also expanded and refined our explanation of how our technique handles private method calls and private variables.

We also added several assumptions, in subsection 1.6.6, that were not mentioned in our previous paper. Some were added to better reflect the core Java constructs that we will be using in our soundness proof in Chapter 4. For example, for simplicity, we are not going to consider inner types, exceptions, array variables, static fields and methods, or package visible fields and methods. We require that the superclass variables and the superclass invariant be visible to subclasses to simplify the application of our rules. Also, we eliminated some assumptions that we thought would be better handled in the rules given in Chapter 3 for preventing unexpected side-effects.

Finally, we need to be able to reason about superclass behavior from method specifications and without the code. However, we found that the modularity of our reasoning technique requires some way of controlling aliasing and preventing unexpected side-effects. The next chapter describes how we prevent the aliasing that could make our technique unsound or non-modular.

CHAPTER 3: PREVENTING UNEXPECTED SIDE-EFFECTS

3.1 Introduction

In our technique, clients reason about the behavior of methods and fields of an object using its public specification, i.e., using the specifications of public methods with respect to the public model fields of the type. The value of the abstract data structure composed only of model fields, as declared in an object's type specification, is the *abstract value* of that object. Therefore, to reason modularly and soundly about the behavior of methods and fields, the abstract value of an object must not change unless the specification allows it to change.

In Section 1.5, we gave examples of some of the kinds of reasoning problems that can arise due to aliasing. In particular, when an internal object is aliased, it can be accessed and modified through other perhaps unrelated objects or variables. Thus unexpected changes to the perceived state of the enclosing object can occur when different objects are making conflicting or unexpected changes to the same aliased object. For example, we showed that modification of an aliased object can inadvertently change the abstract value of other objects in the system which can sometimes cause a type invariant to become invalid.

Furthermore, modular reasoning may not be possible because proofs may require knowing about all the potential and actual aliases in the system, and thus proofs may also require reasoning about the states of objects in unrelated classes in different modules. Also, determining all potential aliases may not be possible, in general, because the aliases created will depend on the run-time execution of the program, and thus the checking of side-effects on all possible aliased objects may not be computable precisely and statically.

Therefore, modular reasoning requires that the abstract value of an object change only if the specification of a called method allows it to change. That is, no mutable concrete field that determines the abstract value of any object in the system can be allowed to change unexpectedly. Thus these mutable concrete fields need to be protected. We say that a concrete field is *mutable* if there are methods (and hence types containing such methods) that can assign to that field.

Concrete fields of the receiver are protected via the assignable clause and our assumption that methods are only allowed to directly assign to fields of the receiver (see subsection 1.6.6). However, if mutable fields of internal concrete objects are related to the value of a model field, then our technique must also prevent these internal objects from being modified unexpectedly. For example, the value of a model field must not be derived from mutable fields of internal objects if, because of aliasing, those fields can be changed by unrelated classes or methods of external objects.

There are two general approaches to protecting the state of internal objects, either disallow the sharing of these objects or allow sharing using read-only references [BNR01]. An object o is *shared* if more than one object contains a reference to o. Also, by definition the state of an object cannot be modified through a *read-only reference*.

In the first approach, unrelated objects are not allowed to contain references to the unshared object; thus the state of an object is encapsulated by encapsulating references to that object [Hog91, Alm97, VB01]. One way this can be accomplished is through unique variables; a *unique variable* is the only variable allowed to contain the reference to an unshared object. Techniques that use unique variables also incorporate a type system and sometimes other mechanisms (such as destructive reads) that are used to guarantee this uniqueness property [Bak95, Min96, Boy01, AKC02, LPHZ02]. Other variations of this approach use ownership rather than uniqueness to increase flexibility [CPN98, NVP98, CNP01, CD02, CW03] but all are based on the same principle of reference encapsulation.

In contrast, the second approach allows the sharing and aliasing of internal objects through read-only references [HLR+99, Mül01, MPH01, KT01, Sko02, LP06]. However, modular reasoning is not possible unless modification of an internal object is not allowed except via fields of the encapsulating object. Thus this second approach must encapsulate side-effects rather than object references.

Our technique uses this second approach since it seems to be less restrictive in the programs allowed. However, unlike the others, our technique only uses type specifications and requires no additional annotations. The specification is used to determine which internal objects need to be protected (Sections 3.3 and 3.4) so our tool can make sure these objects are properly protected through a simple static, modular check (Section 3.5). Thus our technique uses specifications to prevent unexpected and unwanted side-effects while allowing sharing and aliasing of internal objects.

Preventing these unwanted side-effects is important in our technique because we need to be able to reason about superclass methods from the specifications. That is, we have to be able to prove correctness of subclass methods without the superclass code, i.e., using only the superclass specifications, subclass specifications, and subclass code. Thus it is important that client programs and subclass code not inadvertently cause mysterious or unexpected changes to the abstract value of an object.

There are three main goals of our technique for controlling side-effects on internal objects. The first is that it must prevent the side-effects that could make our reasoning system unsound or non-modular. The second is that our technique must be statically checkable[1]. The third goal is that it must have minimal notational overhead so it is easy to use and practical for specifiers and customizers.

We also extend in several significant ways the previous techniques for controlling side-effects. In the current literature, data groups [Lei98, LPHZ02], abstract fields [Lei95, Mül01, Mül02, LN02], and dependency relationships [Lei95, LN02, Mül01, Mül02] are used to specify and control side-effects. However, unlike these techniques, we allow the represents clause to map concrete object structures to abstract object structures; in contrast, the existing techniques require that the mapping be to a value in the language and not to an abstract object structure. Furthermore, we allow method calls and model fields to be accessed in the represents clause (with some restrictions). We also have rules that allow

1. A secondary goal is that our rules produce a minimal number of spurious warning messages.

the type invariant to access of the state of abstract (model) objects, and we provide rules for handling our layered approach to specification, i.e., public, protected, and private specifications; this includes some special rules for handling private fields. Finally, unlike these existing techniques, we provide rules for handling relationships between superclass fields and subclass fields in the invariant and represents clauses[2].

The concern addressed in this chapter is the aliasing induced by object references (pointers) in combination with mutable objects. In Section 3.2, we first motivate the need for our technique by defining our terminology as it relates to the problems caused by aliasing and mutable objects. Then Sections 3.3 and 3.4 describe our rules for determining the objects that must be protected, and Section 3.5 then provides the rules that protect the state of those objects against unexpected side-effects. Finally, Section 3.6 discusses the limitations of and what is accomplished by our technique; it also gives directions for future work.

3.2 Terminology and Concepts

This section defines the terms and concepts used to describe the problems caused by aliasing of mutable objects; these terms are also used in the explanations of our rules for preventing unwanted side-effects.

3.2.1 Static vs. Dynamic Aliasing

An object is *aliased* in a given state if there are at least two access paths to it. An *access path* is a sequence of variable or field names with each name denoting a context (i.e. an object or class) in which the succeeding name is evaluated. For example, in Java, a sequence of names separated by a dot (.), i.e. a qualified name, specifies an access path for an object[3]. When an object can be referenced by more than one qualified name, ignoring the language's visibility restrictions, then each name (or access path) is an *alias* for the object.

When the first name in an access path is either a static field or an instance field, then that path is a *static access path*; however, if the first name is a stack-based variable, i.e., a local variable or parameter, then it is a *dynamic access path*. Thus there are two types of aliases: static and dynamic. A static access path to an aliased object is a *static alias*, whereas a dynamic access path to that object is a *dynamic alias*.

Static and dynamic aliases are handled differently because static aliases can persist beyond method calls, whereas dynamic aliases do not. For example, when an object (reference) is passed as an

2. In Chapter 7, we give a more complete review and comparison with the other alias control techniques in the current literature.

3. This notation applies to languages like Java or Eiffel, but not always to C++ where references to objects (pointers) are not implicit. In C++, the names in an access path are separated by –> rather than a dot when the names denote a reference (pointer) to an object. Smalltalk has no syntax for qualified names, but still has the concept of an access path.

argument in a method or constructor call, then a dynamic alias is created. Dynamic aliases are temporary, because the prefix of the dynamic access path is a local variable or parameter that goes out of scope; that is, dynamic aliases disappear at the end of the execution of the method or block in which they are declared. A static alias occurs, for example, when two or more different fields reference the same object; the fields can be from the same or different objects. Thus a static alias persists until one of the objects containing a field in its access path is no longer accessible (e.g., can be garbage collected) or until one of the fields is assigned a new value.

3.2.2 Abstract Data Types and Representation Exposure

In an object-oriented language like Java, the implementation of an abstract data type is given by a class definition. The benefit of abstract data types is that they can be instantiated and reused in different contexts. To support abstraction, field declarations have access modifiers, such as protected and private, that restrict access to internal fields and objects. We say that an object *I* is *internal* to object *O* if they are different objects and there is an access path from *O* to *I*.

However, restricted access alone is not sufficient to prevent the aliasing and side-effects that could cause incorrect or unexpected behavior. For example, unwanted aliasing can also occur with representation exposure. *Representation exposure* means allowing unrelated objects or unrelated classes to have knowledge of or direct access to the internal implementation of an abstract data type. Recall that two *classes are unrelated* if neither is a subtype of the other. On the other hand, two *objects are unrelated* if neither object is internal to the other. Thus two objects can be unrelated even if they have related types. For example, in most cases, two objects with the same type will be unrelated. For example in Figures 3.1-3.2, two objects with type `CellPair` cannot be related since neither object can be internal to the other, i.e., none of the reachable fields of a `CellPair` object can reference another `CellPair` object (at least with the classes and subclasses of `IntCell` we have seen so far).

Preventing representation exposure involves hiding an abstract data type's representation from unrelated classes and from unrelated objects. Thus there are two ways to prevent representation exposure: information hiding and encapsulation of state. *Information hiding* means that all the internal algorithms, data structures, and classes, used in the implementation of an abstraction, are hidden from unrelated classes. Therefore, different algorithms and representations can be used without affecting clients of the class.

Encapsulation of state, on the other hand, means that an object does not share mutable state with unrelated objects. The *state of an object* includes both the state of its instance variables and the states of its internal objects. An object is *mutable* if clients can change its state during execution of a program. Thus the state of an object is mutable if that object has methods (or calls methods) that assign to any of its fields. However, an object is also mutable if the state of any of its internal objects can be modified by methods of unrelated objects, e.g., when an internal object has a static alias from outside the object.

```
public class CellPair {
   //@ public model IntCell firstCell;
   //@ public model IntCell secondCell;
   //@ public model int secondValue;

   //@ public invariant firstCell != null && secondCell != null;
   //@ public invariant secondValue == secondCell.value;

  /*@  public normal_behavior
    @     requires cellOne != null && cellTwo != null;
    @     assignable firstCell, secondCell;
    @     ensures firstCell == cellOne
    @        &&  secondValue == cellTwo.value;    @*/
    public CellPair(IntCell cellOne, IntCell cellTwo);

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result == firstCell;  @*/
    public /*@ pure @*/ IntCell first();

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result.value == secondValue;  @*/
    public /*@ pure @*/ IntCell second();

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result == secondValue;  @*/
    public /*@ pure @*/ int value2();

  /*@  public normal_behavior
    @     requires newCell != null;
    @     assignable firstCell;
    @     ensures firstCell ==  newCell;  @*/
    public void setFirst(IntCell newCell);

  /*@  public normal_behavior
    @     requires newCell != null;
    @     assignable secondCell;
    @     ensures secondValue == newCell.value;  @*/
    public void setSecond(IntCell newCell);
}
```

Figure 3.1: Public specification of `CellPair` from file `CellPair.jml-refined`.

As we shall see, sharing mutable state does not always cause problems. For example, two objects always share state if one of them is internal to the other; this is usually necessary when building large systems. However, problems can arise when two unrelated objects share state, in particular, when the same mutable object is internal to both. Thus the primary goal of our alias control technique is to prevent such sharing when it makes our reasoning technique unsound or non-modular.

3.2.3 Layers of Abstraction

Abstract data types are also used to create layers of abstraction. Modern computing systems are built in layers of abstraction; this is one of the most important methods for managing the complexity of large systems. For example, in a large system of Java classes, objects at a *higher level* of abstraction are implemented using objects at lower levels. Higher-level objects are typically more abstract, whereas *lower-level* objects are more detailed, specific, and concrete. Also, higher-level abstractions are usually closer to end user needs.

A similar phenomenon occurs in specifications when higher-level objects are used to specify the public behavior of an abstract data type, while lower-level objects are used in its concrete implementation. For example, in JML, a public model field is used to specify the behavior of a group of one or more concrete fields. The abstract value of that model field is a function of the values of the concrete fields in this group; in JML, each model field can have a represents clause that specifies this mapping (or relation). Also, since changes to these concrete fields affect the value of a model field, they must all be members of that model field's data group. For example, in Figures 3.1 and 3.2, the public model field `firstCell` is implemented using the concrete field `_first`, and `_first` is a member of `firstCell`'s data group; similarly `secondValue` is implemented using `_second`, which is a member of `secondValue`'s data group.

Model fields are also used to hide implementation details, that is, concrete fields are not usually visible to clients but model fields are[4]. For example, clients see only the public specification in Figure 3.1. This public specification of `CellPair` is an example of a higher-level abstraction. Figures 3.2 and 3.3 are examples of lower-level abstractions, i.e., the protected specification and concrete implementation respectively give the implementation details.

3.3 Pivot Fields

In this section, we describe how our technique prevents the side-effects and aliasing that could make reasoning unsound or non-modular. Our technique uses this notion of layers of abstraction to accomplish information hiding, i.e., hiding the implementation details from client programs. However, the soundness of our technique also depends on protecting the state of concrete objects used to represent and determine the abstract value of an object. Thus our technique has to provide rules for determining which objects need to be protected from unexpected side-effects and rules that ensure that

4. Our technique requires that concrete fields not be public, i.e., not be directly accessible by clients.

```
//@ refines "CellPair.jml-refined";

public class CellPair {

    protected IntCell _first;
    //@                in firstCell;

    protected IntCell _second;
    /*@                in secondCell, secondValue;
      @                maps _second.value \into secondValue;
      @*/

    protected int _val2;
    //@             in secondValue, secondCell;

    //@ protected represents firstCell <- _first;
    //@ protected represents secondCell <- _second;
    //@ protected represents secondValue <- _second.value;

    //@ protected invariant secondValue == _val2;
}
```

Figure 3.2: Protected specification of `CellPair` from file `CellPair.jml`.

these objects are not modified by methods of unrelated objects. We call the fields that reference these protected internal objects pivot fields. Our technique does not encapsulate these objects, but rather it only allows the owner of these pivot objects to modify them. In particular, each pivot field owns (or is the owner variable[5] of) the pivot object it references and therefore must be the only variable in the system allowed to initiate changes to the state of that pivot object.

3.3.1 Background and Definitions

A *pivot field* [LS99, LN02] is a concrete instance field that references a lower-level mutable object and some part of the state of this lower-level object is accessed by a model field or a predicate clause declared in the specification of the type. We will also refer to the object referenced by a pivot field as a *pivot object*. Furthermore, an object will be referred to as object *x* when it is referenced by a field named *x*.

The values of model fields are also constrained by predicate clauses. A *predicate clause* is a clause in a type specification that makes a boolean assertion about the state of objects in that type, such as, the invariant, requires, and ensures clauses. Because clients need to reason about the value of the public

5. Subsection 3.5.1 more specifically defines what we mean by an owner variable.

```
//@ refines "CellPair.jml";

public class CellPair {

    protected IntCell _first;
    protected IntCell _second;
    protected int _val2;

    public CellPair(IntCell cellOne, IntCell cellTwo) {
        _first = cellOne;
        _second = cellTwo;
        _val2 = _second.getValue();
    }
    public /*@ pure @*/ IntCell first() {
        return _first;
    }
    public /*@ pure @*/ IntCell second() {
        return _second;
    }
    public /*@ pure @*/ int value1() {
        return  _first.getValue();
    }
    public /*@ pure @*/ int value2() {
        return  _val2;
    }
    public void setFirst(IntCell newCell) {
        _first = newCell;
    }
    public void setSecond(IntCell newCell) {
        _second = newCell;
        _val2 = _second.getValue();
    }
}
```

Figure 3.3: An incorrect implementation of `CellPair` from file `CellPair.java`.

model fields of a type, the mutable concrete fields indirectly accessed by a predicate clause must also be protected from unexpected change. For example, if a concrete field is related to the type invariant via a model field and represents clause, then that field must be protected so the invariant is not inadvertently invalidated.

In JML, the state of a concrete pivot object is either related to a higher-level abstraction, e.g., a model field, through a represents clause or to the value of a literal or another model or concrete field in a predicate clause. For example, in Figure 3.2, concrete field `_second` is a pivot field because part

of the state of the object it references, the indirectly declared field[6] `_second.value`, determines the abstract value of model field `secondValue` in the higher-level abstraction. The specific mapping (or relation) between the lower and higher-level abstractions is given in the represents clause.

The state of a mutable object can also be accessed in a predicate clause. The *state of an object x is accessed* when one of *x*'s concrete fields is (directly or indirectly) accessed; that is, accessing an indirectly declared field, such as *x.V*, means accessing the state of object *x*.

Unrelated objects must not be allowed to change an internal object whose state is accessed by a predicate clause. For example, if an invariant clause accesses the state of an internal mutable object, then that internal object must be a pivot because this invariant must hold prior to and after method calls. Furthermore, when the pre- and postconditions in the requires and ensures clauses, respectively, make assertions about the state of a concrete internal object (usually through model fields), then this object must be a pivot because clients are, in effect, reasoning about the state of that concrete internal object.

Similarly, unrelated objects must not be allowed to change concrete fields accessed by a represents clause because such changes may change the abstract value of a model field. For example, the second invariant clause in Figure 3.1 specifies a relationship between `secondValue` and `secondCell.value`. In this case, the invariant relationship is specified using model fields but pivot fields are always concrete fields. Therefore, if the state of a concrete object is accessed in the represents clause of either of these model fields, then that concrete object must be a pivot. Hence, `_second` in Figure 3.2 must be a pivot field since the state of the object `_second` is accessed by the represents clause of `secondValue`.

However, `_first` is not a pivot field because the state of the object it references is not accessed by a predicate clause and it does not determine any part of the abstract value of a `CellPair` object. That is, in Figures 3.1 and 3.2, none of the represents, invariant, requires, and ensures clauses in `CellPair` access the fields of `_first`. Such non-pivot objects can be shared because changing their state cannot invalidate the invariant of a `CellPair` object nor does it change the abstract value of the higher-level abstraction.

Identifying the pivot objects internal to a type is important because the state of these mutable objects must be protected, i.e., methods of unrelated classes and objects must not be allowed to change them since such changes could cause unexpected behavior. For example, method `second` in the implementation of `CellPair` shown in Figure 3.3 returns the pivot object referenced by `_second`. Thus clients can capture references (create aliases) and later change the state of that pivot object, which could invalidate `CellPair`'s invariant. Furthermore, when the state of object `_second` is changed unexpectedly, this changes the state of the abstract value of `secondValue` which could also

6. Recall that instance field *x.V* is *indirectly declared* in type *C* if *x* is directly declared in *C* and *x* references an object with a (directly or indirectly declared) field *V*.

```
public class CellPairClient {

    public static void main( String[] args ) {
        CellPair pair1 = new CellPair(new IntCell(3),
                                      new IntCell(7));
        IntCell c1 = pair1.first();
                   // (1) c1 is an alias of pair1._first

        IntCell c2 = pair1.second();
                   // (2) c2 is an alias for pair1._second

        IntCell c3 = new IntCell(5);
        CellPair pair2 = new CellPair(new IntCell(3), c3);
                   // (3) c3 is an alias for pair2._second

        c1.setValue(4);
                   // (4) invariant of pair1 is OK

        c2.setValue(3);
                   // (5) invariant of pair1 is invalid

        c3.setValue(3);
                   // (6) invariant of pair2 is invalid
    }
}
```

Figure 3.4: Client program of `CellPair`.

cause client programs to go wrong. The client program in Figure 3.4 illustrates how aliasing can cause unexpected problems; we will use this example later to explain these problems and how our rules prevent them.

Thus we need some way of declaring pivot fields so implementers and our tools can recognize them. Furthermore, for soundness, we have to require that all pivot fields be declared; the rest of this section describes how our technique accomplishes this.

3.3.2 Declaring the Pivot Fields of a Type

Our technique uses the maps clauses in field declarations to specify that a field is a pivot. Thus no additional annotations or specifications are needed because these maps clauses are already required for specifying and controlling side-effects. That is, the maps clause is already needed in field declarations to allow side-effects to internal objects. For example, the maps clause in Figure 3.2 declares field `_second` to be a pivot field and allows side-effects to the indirectly declared concrete

fields in data group `value` of field `_second`. Our next rule specifies when a field must be declared as a pivot field, i.e., when a field declaration must also include a maps clause.

Pivot Declaration Rule. Let *x.V* be an instance field indirectly declared in type *C* or one of *C*'s supertypes. If *x* is a concrete field and *x.V* is accessed on the right-hand side of a represents clause or in a predicate clause in the specification of *C*, then *x.V* must be a member of (i.e. mapped into) a data group visible to *C*.

The above rule says that field *x* must be a pivot field whenever the state of a concrete object *x* (which cannot be the receiver) is accessed in a represents or predicate clause since adding an indirectly declared field like *x.V* to a data group requires a maps clause. For example, indirectly declared fields, like `_second.value`, can only be added to a data group through a maps clause as shown in Figure 3.2. Furthermore, this maps clause adds the members of data group `_second.value` to `secondValue`; thus `_second._val` is also added to this data group making `_second` a pivot field.

A maps clause must also be declared to allow any necessary side-effects to these internal objects. However, enforcement of the assignable clauses does not guarantee that all the necessary maps clauses have been declared for pivot fields. For example, the implementation of `CellWrapper` in Figure 3.5 does not require the maps clause shown in that Figure. That is, the state of object `_cell` (i.e., concrete field `_cell._val`) is indirectly updated by method `set` by assigning a new object reference to `_cell`. Nonetheless, `_cell` must be a pivot field because its state is accessed in the represents clause for `theValue`. Furthermore, pivot objects must be declared even if none of their fields are changed by methods of the type (see also the example of a pure type given in Section 3.5). Thus declaring the maps clauses needed to control side-effects is not sufficient to guarantee that all pivot fields have been declared.

In summary, there are two reasons that an internal object needs to be a pivot. First, if methods need to modify the state of an object that is internal to the receiver, then that internal object must be a pivot in the receiver. Second, an internal object has to be a pivot if a represents or predicate clause accesses its state. The second is necessary because, when an internal object of the receiver determines any part of the abstract value of the receiver object, then our technique has to protect the state of that internal object, i.e., prevent it from being changed by methods of other perhaps unrelated classes and objects.

3.3.3 Determining the Concrete Fields Accessed in an Expression

We need to know how the fields accessed by the represents and predicate clauses can be calculated so our tool and technique can ensure that all pivot fields have been declared, i.e., so our tool can apply the above Pivot Declaration Rule to determine whether a maps clause is needed in a field declaration. This subsection describes an algorithm and the necessary syntactic restrictions on represents and predicate clauses so our tool can check that all pivot fields have been declared in a type.

```
public class CellWrapper {

    //@ public model int theValue;

    /*@ public normal_behavior
      @    requires c != null;
      @    assignable theValue;
      @    ensures theValue == c.value;
      @*/
    public void set(IntCell c) {
        _cell = new IntCell(c.getValue());
    }

    ...

    //  Protected Specification below

    protected IntCell _cell;
    //@               in theValue;
    //@               maps _cell.value \into theValue;

    //@ protected invariant _cell != null;
    //@ protected invariant _cell.value == theValue;

    //@ protected represents theValue <- _cell.value;

}
```

Figure 3.5: Public and protected specifications combined with the implementation of part of class `CellWrapper`.

To ensure that the pivot fields have been declared, we have to first calculate the concrete fields accessed in the expressions occurring in represents and predicate clauses since these are the fields that determine the abstract value of an object, i.e., the abstract value of its model fields and of the predicate clauses that constrain the values of these model fields. Thus we need an algorithm for determining which concrete fields are accessed in an expression; based on this algorithm, our tool can determine which indirectly declared concrete fields are accessed by the represents and predicate clauses of a class, and thus which fields have to be pivots.

In general, we say that a field is *accessed* in an expression if it is either directly or indirectly accessed. A visible field, *V*, is *directly accessed* in an expression *E* if *V* occurs in *E*. However, *V* is *indirectly accessed* in *E* if a model field *F* is directly accessed in *E* and *V* is accessed (directly or indirectly) on the right-hand side of *F's* represents clause. *V* is also indirectly accessed in *E* if a method or constructor *M* is called in *E* and *V* is accessed (directly or indirectly) by the call of *M*. Also,

whenever field *V* is accessed in expression *E*, and *V* references an object, then we say that *object V is accessed* in *E*.

Figures 3.1 and 3.2 provide a simple example; that is, `secondValue`'s represents clause, in Figure 3.2, directly accesses `_second.value`, and thus indirectly accesses `_second._val` (since `_val` is directly accessed by `value`'s represents clause in class `IntCell` of Figure 1.5). Thus, the expression `secondValue` indirectly accesses concrete field `_second._val`.

Figures 3.6 and 3.7 formalize the function *accessed*(*E*); this function computes the set of concrete fields accessed in expression *E*. That is, Figure 3.6 defines the function *accessed*(*E*) which calls the auxiliary function, *replacePrefix*, defined in Figure 3.7. Function, *accessed*(*E*), takes an abstract syntax tree (AST) node as its input argument. However, we assume that the type checker has been run, and thus this AST node contains the attributes needed by the function[7]. For example, if type checking has already been done, then the variable and method names have been resolved. Therefore, whether the variable, in rules 1-4, is a field of the receiver, a parameter, a local variable, or a model field can be determined from attributes in this AST; thus, using these attributes, this function can determine which of rules 1-4 should be applied[8]. Specifically in JML, an expression that references a variable will have, as an attribute, a reference to the declaration of that variable.

Furthermore, as required by rule 3, a model field's represents clause can be obtained through any AST node that references that field. Rule 3 says that the concrete fields indirectly accessed through a model field *v* are exactly those fields accessed by *v*'s represents clause.

Notice also that rule 1 always includes the implicit "`this.`" in the result set to avoid ambiguities with parameter names. Also, rules 5 and 7 handle the case where the prefix "`this.`"[9] is explicit in an expression.

Our algorithm also handles method calls in expressions. Rules 2 and 6, taken together, define how the concrete fields indirectly accessed through a method call are determined, and, in particular, how we handle parameters. Since parameters are not concrete fields, parameter references in the body of a method must be converted into the concrete fields indirectly accessed during a method call; this is done, as specified in rule 6, using the actual parameters from the method call expression. That is, for method calls, the actual parameters are, in effect, substituted for the formal parameters in the method

---

7. After the type checker has been run, in Iowa State University's JML tool, the information needed by this algorithm is available or can be found via attributes of the AST.
8. We do not include static fields in the algorithm because, for simplicity, we are assuming that types do not declare static fields (see assumptions in subsection 1.6.6). A static field presents a unique set of problems because it is shared by all objects of a type, i.e., it acts as if it were aliased. Therefore, we consider it an error if a represents or predicate clause accesses such fields (see also the discussion in subsection 3.6.3).
9. In Java, the prefix could also be "`super.`" for accessing superclass fields of the receiver, but, for simplicity, we will not consider that here. Also, in C++, the prefix could be "*C*`::`" where *C* is the superclass name where the field was declared.

1. *accessed*(*v*) = { `this.`*v* } if *v* is a concrete field of the receiver
2. *accessed*(*v*) = { *v* } if *v* is a parameter
3. *accessed*(*v*) = *accessed*(*R*) if *v* is a model field and *R* is the expression on the right-hand side of *v*'s represents clause
4. *accessed*(*v*) = { } if *v* is a local variable other than a parameter
5. *accessed*(`this`) = { `this` }
6. *accessed*(*M*(R_1, ... , R_n)) = *accessed*(R_1) ∪ ... ∪ *accessed*(R_n) ∪ *replacePrefix*(R_1, p_1, ... *replacePrefix*(R_n, p_n, $accessed_M$)...), where $accessed_M$ is the set of fields accessed by method *M* (determined from *M*'s accessible clause)
7. *accessed*(*X*`.`*R*) = *accessed*(*X*) ∪ *replacePrefix*(`this`, *X*, *accessed*(*R*)) if *X* is a concrete access path (i.e., a sequence of concrete variable or parameter names) denoting object *X* and *R* is a variable access or method call expression (as in rules 1 - 6)
8. *accessed*(*X*`.`*R*) = { } if *X* is not a concrete access path or *R* is not a variable access or method call expression
9. *accessed*(*E1* `binop` *E2*) = *accessed*(*E1*) ∪ *accessed*(*E2*)
10. *accessed*(`unop` *E1*) = *accessed*(*E1*)

Figure 3.6: Summary of the rules for determining the concrete fields accessed in an expression; `binop` and `unop` denote the binary and unary operators respectively. Also, *E1* and *E2* denote expressions.

1. *replacePrefix*(*v*, *R*, $accessed_E$) = $accessed_E$ if *R* is not an access path
2. *replacePrefix*(*v*, *R*, $accessed_E$) = { } if $accessed_E$ is empty
3. *replacePrefix*(*v*, *R*, $accessed_E$) = { *R* } ∪ *replacePrefix*(*v*, *R*, $accessed_E$ - { *v* }) if *v*∈$accessed_E$ and *R* is an access path
4. *replacePrefix*(*v*, *R*, $accessed_E$) = { *R*`.`*Y* } ∪ *replacePrefix*(*v*, *R*, $accessed_E$ - {*v*`.`*Y*}) if *v*`.`*Y*∈$accessed_E$ and *R* is an access path
5. *replacePrefix*(*v*, *R*, $accessed_E$) = { *w* } ∪ *replacePrefix*(*v*, *R*, $accessed_E$ - { *w* }) if *w*∈$accessed_E$ and *w* ≠ *v*
6. *replacePrefix*(*v*, *R*, $accessed_E$) = { *w*`.`*Y* } ∪ *replacePrefix*(*v*, *R*, $accessed_E$ - { *w*`.`*Y* }) if *w*`.`*Y*∈$accessed_E$ and *w* ≠ *v*

Figure 3.7: Definition of auxiliary function *replacePrefix*; it replaces the prefix *v* in all paths in $accessed_E$ with the access path *R*. In this algorithm, *v* is either the receiver `this` or a parameter name.

body. Thus the concrete fields accessed by the call are determined, as described in rule 6, by taking a method's accessible clause, i.e., $accessed_M$, and replacing the occurrences of the formal parameters with the actual parameters using the *replacePrefix* function. Chapter 6 describes how our tool calculates the fields accessed in the body of a method (we use this same algorithm for determining the fields accessed in expressions and assignment statements, and then union together the fields accessed in each statement in the body).

In addition, our algorithm needs to handle qualified names. Rule 7 defines how to determine the access paths referenced by a qualified name[10]. In rule 7, the set, *accessed*(*R*), has to be computed first, because the subexpression *R* can (indirectly) access concrete fields through a model field or a method call. We then have to add the prefix *X* to the appropriate access paths in *accessed*(*R*) using *replacePrefix*. That is, we must only replace the prefix of a path in *accessed*(*R*) if the prefix is "`this.`", since the prefix of any other access paths must necessarily reference a parameter; that is, they do not reference the receiver *X*. Furthermore, rule 7 requires that the prefix, *X*, be an access path that only includes concrete field or parameter names. This is because we want access paths in *accessed*(*X*`.`*R*) to be static access paths or paths that can be converted into a static path; thus access path *X* cannot reference model fields or local variables (other than parameters). Subsection 3.3.4 explains why model fields are not allowed in *X*. We do, however, allow parameter names to be part (the prefix) of concrete access paths because they can and must eventually be converted, by application of rule 5, into paths consisting solely of concrete fields; rule 8 says to ignore the other access paths. Similarly, if the actual parameters in a method call are not access paths that can be converted into a concrete path, then the algorithm ignores them (see rule 7 of *accessed* and rule 1 of *replacePrefix*).

Rules 9 and 10 are included to show how subexpressions are handled, i.e., the fields accessed in subexpressions are unioned together. These two rules do not represent all the possible expressions and combinations of expressions in Java. Nonetheless, the included rules are meant to illustrate that all fields accessed in subexpressions are unioned together.

However, some expressions do not access any concrete fields (e.g., literals). Furthermore, there are expressions for which the fields accessed cannot, in general, be precisely calculated (e.g., fields of model objects, as explained in subsection 3.3.4); rules 4 and 8 are also examples. Therefore, primitive expressions, other than the variable references handled by rules 1-3 and 5-7, do not add concrete fields to *accessed*(*E*).

Finally, we believe that this algorithm handles most practical situations. However, when an expression is not handled (e.g., a field of a model object), then the specification can, in most cases, be changed to something that can be handled (see subsection 3.3.4).

10.For simplicity, we do not include arrays in the rules given in Chapters 2 and 3. However, we believe that our technique can be extended to handle arrays (see assumptions in subsection 1.6.6).

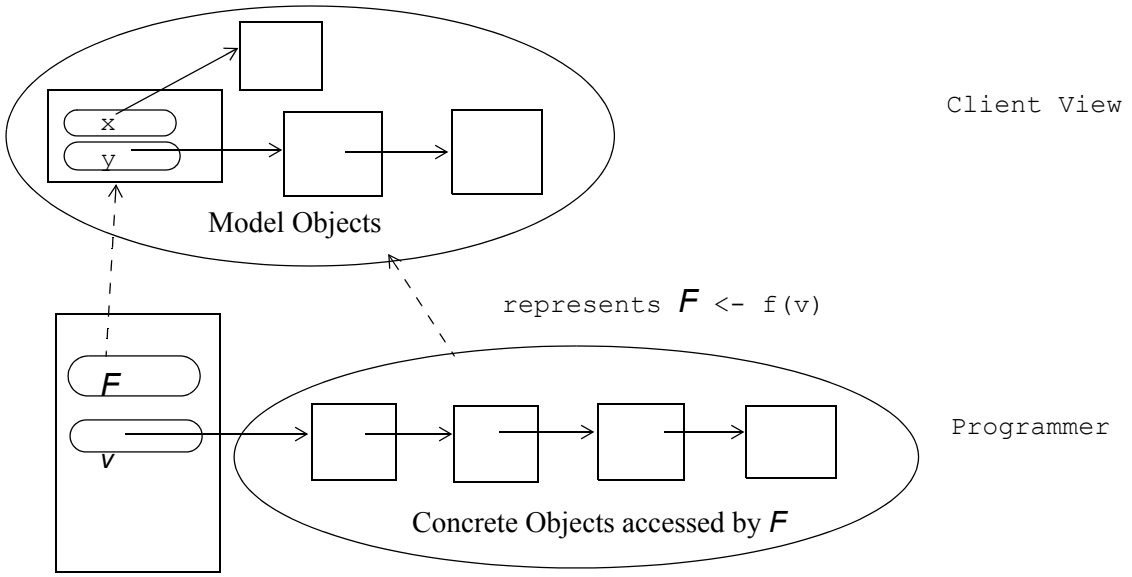

Figure 3.8: Concrete objects referenced through field *v* are mapped to the model objects referenced by *F*.

3.3.4 Accessing Fields of Model Objects

The algorithm, in Figures 3.6 and 3.7, calculates the concrete fields accessed by an expression; however, it does not specify how to determine the concrete fields accessed by fields of model objects. The problem related to model objects is that these objects are abstract, i.e., they do not exist during program execution. Therefore, all fields of a model object are also abstract in the sense that they also do not exist during program execution. Thus when a field of an abstract object is accessed, it is not always obvious which concrete fields are indirectly accessed by such expressions. This subsection and subsection 3.3.5 explain when and how our technique deals with fields of model objects, and why we restrict the syntax of the predicate, represents, and assignable clauses to expressions that have a precise meaning and that our tool can handle.

The state of a model object is computed from a represents clause. Therefore, a field of a model object indirectly accesses the concrete fields used to compute its value. For example, suppose a model field *F* references an object as shown in Figure 3.8; object *F* is mapped from the concrete objects accessed through field *v*. However, because object *F* has no concrete state or concrete fields, we have to consider that an expression like *F.x* indirectly accesses all fields in *accessed*(*F*)[11], since they determine the abstract value of both *F* and *F.x*. For example, in Figure 3.9, the model field `cell` is determined by `_pair._first` through its represents clause and the represents clause for

11.The value of *accessed*(*F*) is determined from the represents clause for *F* as shown in Figure 3.6.

`firstCell` in Figure 3.2. However, `_pair._first` is not a pivot field; this is fine as long as the type invariant and abstract value of `BadInvariant` objects do not access fields of `cell`. However, the second invariant clause requires that `cell.value` be greater than or equal to zero. Therefore, this invariant cannot be allowed because `_pair._first` can be aliased which could allow clients or methods of unrelated classes to invalidate this invariant. Our next rule disallows references to fields of model objects in predicate clauses when that model object depends on non-pivot, concrete fields.

Predicate Clause Access Rule. Let class *S* be *C* or a subtype of *C*. Let *F* be a model field with a reference type that is directly declared in *C*. If *F*'s represents clause accesses a non-pivot object *V*, then fields of model object *F* cannot be accessed by predicate clauses in *C* and *S*.

This rule prevents aliasing problems that can arise when a model field accesses a non-pivot field. For example, it disallows the second invariant clause in Figure 3.9, as required, because it accesses `cell.value` and `cell` indirectly accesses `_pair._first` which is not a pivot field.

However, there is another more difficult problem that this rule does not handle. For example, if we look again at Figure 3.8, the value of field *F.x* could also be determined by concrete fields that are not in *accessed*(*F*). In particular, *F.x* may depend on fields of objects referenced by members of *accessed*(*F*). For example, in Figure 3.9, the represents clause for `pair` specifies that it references the same object as `_pair`. Thus `_pair._first` and `_pair._first._val` are indirectly accessed by the first invariant clause in Figure 3.9. However, neither of these concrete fields are accessed by `pair`'s represents clause. Therefore, the concrete fields accessed by the represents clause are not necessarily the fields that determine the value of any particular field of a model object.

Nonetheless, in Figure 3.9, we were able to easily determine which concrete fields were indirectly accessed by expression `pair.firstCell.value`. In general, however, it may not be so simple to determine from the represents clause which concrete fields are mapped to the value of a particular field of a model object, i.e., the specified mapping may be complicated or it may not be one-to-one. Thus, to be safe, we have to assume, in general, that when *F* references a model object, then *F.x* could depend on the value of some fields of every concrete object reachable through *accessed*(*F*). Thus, when *F.x* is accessed in a represents or predicate clause, then every object reachable through *accessed*(*F*) would have to be a pivot. Class `BadInvariant` in Figure 3.9 illustrates the problem. That is, `_first` is not a pivot in `CellPair` so our technique allows `_pair._first` to be aliased; if it is aliased, then the first two invariant clauses in Figure 3.9 could be invalidated by its owner since both `cell.value` and `pair.firstCell.value` depend on the state of `_pair._first`. Thus, unless the fields accessed can be precisely calculated and checked, then all objects reachable through `_pair` would have to be pivots.

However, we do not believe this to be a practical approach, in general, because it would unnecessarily require declaring internal objects as pivots. For example, one would not be able to specify a list of sharable objects if the list nodes are accessed by the represents clause of a model

```
public class BadInvariant {

    //@ public model CellPair pair;

    /*@ public model IntCell cell;
      @                      maps cell.value \into cell;  // not allowed
      @*/
    //@ public model IntCell cell2;

    /*@ public invariant pair != null
      @               && pair.firstCell.value >= 0;     // not allowed
      @*/
    //@ public invariant cell != null && cell.value >= 0;  // not allowed
    //@ public invariant cell2 != null && cell2.value >= 0;

// protected specification follows:

    protected CellPair _pair;
    /*@                in pair;
      @                maps _pair.firstCell \into cell;
      @                maps _pair.secondCell \into cell2;
      @*/

    //@ protected represents pair <- _pair;
    //@ protected represents cell <- _pair.firstCell;
    //@ protected represents cell2 <- _pair.secondCell;
}
```

Figure 3.9: Combined public and protected specification of `BadInvariant` from file `BadInvariant.jml`.

field *F* and a field *F.x* is accessed by the invariant. With such a restrictive rule, it would be much more difficult to share non-pivot objects and to create reusable types.

Therefore, in JML, we want to allow predicate clauses that access the state of model objects when it is sound to do so, i.e., when the predicate does not depend on a reachable non-pivot object. For example, we would like to allow expressions such as `cell2.value` in the third invariant clause of Figure 3.9, but without requiring that all objects reachable through *accessed*(`cell2.value`) be pivots, i.e., without requiring that all objects reachable through `_pair` be pivots (the right side of `cell2`'s represents clause accesses `_pair`). That is, because `cell2.value` does not depend on any non-pivot fields of `_pair`, i.e., it does not depend on `_pair._first`, this invariant should be allowed.

However, as described above, it may not be possible for a tool to automatically determine the pivot fields in all situations. That is, the more complicated cases may require a hand crafted proof to ensure

that all pivot fields of internal objects have been declared (and this would not always be a trivial task). Nonetheless, determining the dependencies is straightforward when a concrete field and a model field have the same type and reference the same object, as in the represents clause for `secondCell` in Figure 3.2. In such cases, there is a one-to-one correspondence between the fields of the model object and the concrete field. Thus our next rule allows the state of such model objects to be accessed in predicate clauses.

Model Field Access Rule. Let class *S* be *C* or a subtype of *C*. Let *F* be a model field with a reference type that is directly declared in *C*. If *F*'s represents clause maps a concrete field *V* to *F*, i.e., if they both denote the same object, then fields of model object *F* can be accessed by predicate clauses in *C* and *S*, otherwise fields of *F* cannot be accessed by predicate clauses.

The above rule allows fields of a model object *F* to be referenced, for example, in an invariant clause when *F* and a concrete field *V* denote the same object. However, if *V* is not a pivot, then, as required, fields of *F* cannot be accessed by the invariant based on the above Predicate Clause Access Rule.

In any case, it is usually better to avoid accessing the state of model objects in the invariant; that is, the specifier can always map the concrete field *V.x* to a model field *G* and then reference *G* when necessary, as was done for `secondValue` in Figure 3.2. In either case, i.e., whether *F.x* or *G* is accessed in the invariant, concrete field *V* has to be a pivot. In addition, we allow this exception because it occurs frequently in specifications[12], is easy for a tool to check, and because we want to make it convenient to express invariants in the public specifications whenever it is possible and sound to do so.

However, we do not have to be so restrictive when dealing with the represents clause. That is, if we require that the syntax of expressions in the represents clause conform to the grammar given in Figure 3.10, then our tool can precisely calculate the concrete fields indirectly accessed by a model field, i.e., we can use the algorithm in Figures 3.6 and 3.7. This is possible because, unlike a public predicate clause, the represents clause is given in the protected or private specification where the concrete fields are visible and can be accessed. Therefore, this restricted syntax allows our tool to calculate the pivot fields related to each model field, i.e., the concrete fields indirectly accessed through a model field, which is necessary for the soundness of our technique (notice that there is a rule in the definition of *accessed* in Figure 3.6 that handles each production rule shown in Figure 3.10). Our next rule specifies our restrictions on the represents clause.

---

12. In JML, the spec_public modifier for a concrete field *F* is a short hand for declaring a public model field *F* and a concrete field *V* with the same type, and a represents clause mapping *V* to *F*; furthermore, the methods of the class are, as if, rewritten so the references to *F* are replaced by *V*. The spec_protected modifier is similar except the visibility of the model field is protected.

represents_expr ::= *concrete_field_access*
| *model_field_access*
| *method_call_access*
| *represents_expr binop represents_expr*

concrete_field_access ::= *concrete_field_name*
| *self*
| *self* . *concrete_field_name*

model_field_access ::= *model_field_name*
| *concrete_field_access* . *model_field_name*

method_call_access ::= *method_call*
| *concrete_field_access* . *method_call*

self ::= `this`
| `super`

method_call ::= *method_name* (*argument_list*)

argument_list ::= *actual_parameter*
| *actual_parameter* , *argument_list*

actual_parameter ::= *concrete_field_access*
| *represents_expr binop represents_expr*

Figure 3.10: The allowable syntax for expressions on the right-hand side of represents clauses (`binop` denotes the binary operators in Java).

Represents Clause Access Rule. Expressions occurring on the right side of a represents clause must follow the syntax given in Figure 3.10.

Notice that this rule disallows a field of a model object from being accessed in a represents clause, i.e., the restricted syntax does not allow this[13]. Furthermore, our tool considers it an error if the represents clause accesses fields of a model object. Similarly, it is an error if fields of a non-pivot object are accessed by a represents clause since this means the Pivot Declaration Rule has been violated (see subsection 3.3.2).

3.3.5 Model Fields in the Assignable Clause

Referencing a data field of a model object in an assignable clause has a problem similar to the problems described earlier related to model objects in represents and predicate clauses (subsection 3.3.4). That is, the purpose of the assignable clause is to specify the set of concrete fields that a

13. We leave any additional weakening of the restrictions on expressions allowed in invariant and represents clauses as future work.

method is allowed to assign to; however, the fields of a model object do not specify a set of concrete fields because a model object is abstract and has no concrete state or concrete fields; thus the fields of a model object cannot be assigned to during method execution. Furthermore, as described in subsection 3.3.4, the concrete fields that a field of a model object would denote cannot always be precisely calculated. Therefore, our technique does not allow fields of model objects to be listed in an assignable clause. Furthermore, this restriction is in line with the current techniques for specifying and controlling side-effects through the use of data groups and abstract field dependencies. That is, in the current literature, data groups [Lei98, LPHZ02] and abstract fields [Lei95, Mül01, Mül02, LN02] do not have attributes (they are not object structures), so the state of a data group or abstract field has no meaning; hence, it cannot be directly referenced or changed.

Furthermore, the purpose of the in and maps clauses is to add concrete fields into a data group. Therefore, the same problem exists for a maps clause when it appears in the declaration of a model field. For example, the declaration of model field `cell` in Figure 3.9 has a maps clause. However, data group `cell.value` does not contain any concrete fields because the object referenced by `cell` is abstract. Therefore, JML and our alias control technique do not allow the maps clause in model field declarations.

We do, however, allow the in clause in the declaration of a model field because a model field has an associated data group containing concrete fields; these concrete fields are added to the data groups listed in its in clauses (see Figure 3.17). Our next rule restricts the expressions allowed in an assignable clause and disallows the maps clause in model field declarations.

Assignable clause rule. A field of a model object cannot be directly accessed in an assignable or maps clause.

One might think that we could make an exception to the above rule when the concrete object and model object are the same (e.g., `secondCell` and `_second` in Figures 3.1 and 3.2). However, if this exception were allowed, then an overriding subclass method might not be able to assign to additional subclass fields; thus, in effect, the behavior of some subclass methods would not be specifiable. For example, if `secondCell.value` were the only group listed in an assignable clause, then an overriding method could not assign to additional subclass fields since fields of the receiver cannot be added to groups declared in other objects, e.g., indirectly declared data groups such as `secondCell.value`[14] (see also subsection 3.4.1). Instead model field `secondValue` has to be listed in the assignable clause since that group contains the required concrete field and subclasses can extend that group with additional subclass fields.

Examples given in Chapters 1 and 2 show that it is sometimes necessary to override methods and to assign to subclass fields in order to satisfy subclass method specifications and subclass invariants. In

14. This is needed for soundness [Lei95, Lei98, LPHZ02]; thus the maps clause syntax does not allow a field to be added to a data group in another object.

such cases, subclass fields have to be added to superclass data groups to allow those side-effects; this is the primary purpose and use of data groups (i.e., to allow additional side-effects) and is required by some of the rules given below. Therefore, to maximize extensibility of methods, only public data groups and model fields of the receiver should be listed in public assignable clauses, since otherwise it would not be possible for an overriding subclass method to modify subclass fields when only fields of objects are listed[15].

The above rule also disallows the maps clause in model field declarations, because all maps clauses in the declaration of a field *F* must access a directly declared field or group in object *F*, such as *F*.*x*; the syntax and semantics of the maps clause require this. Therefore, because the above rule does not allow a maps clause to access a field of a model object, there can be no valid maps clauses for a model field *F*.

3.4 Specifying and Controlling Side-Effects

The rules given in this section formalize our technique for ensuring that side-effects are properly controlled and specified. In particular, when the specification allows a method to modify a concrete field, then related fields must also be allowed to change. These rules also ensure that all pivot fields accessed by a represents clause have been declared. Stated another way, these rules, if followed, prevent fields of a non-pivot object from being accessed by a represents clause; this is accomplished by requiring data group memberships that also make sure that the pivot fields have been declared through a maps clause.

3.4.1 The Syntactic Restrictions in Maps and Represents Clauses

The syntactic restrictions in the maps clause are necessary for soundness [Lei95, Lei98, LPHZ02] and are closely related to our restrictions in the represents clause (Figure 3.10). In particular, the maps clause syntax does not allow a field like `x.y.z` to be directly mapped into a data group. This syntactic restriction is important because it forces data group relationships that are necessary for the soundness of our technique[16]. The following example illustrates why this restriction is important for soundness and modularity.

3.4.1.1 An example

Suppose `x` is a concrete field and `y` is a concrete field directly declared in object `x`. Suppose also that expression `x.y.z` (more precisely `this.x.y.z`) is accessed in the represents clause of a

15. Note, however, that this restriction only applies to objects referenced by the receiver, that is, fields or data groups declared in parameter objects can be listed in the assignable clause since such side-effects do not change the state of the receiver (recall from subsection 1.6.6 that we assume that objects referenced by fields of the receiver are not passed as arguments in self-calls). Thus subclasses are only extending the behavior and state of the receiver and not the behavior or state of explicit parameter objects.

16. This is the same syntactic restriction as the one given in Leino *et al.*'s paper [LPHZ02] for the same reasons.

model field *F*. Therefore, to satisfy the Model Field Data Group Rule (subsection 3.4.2), `x.y.z` must be added to data group *F* (or some other group in the receiver). However, in the declaration of field `x`, a field like `x.y.z` cannot be mapped into a data group. That is, the syntax of the maps clause only allows a field or group that has been directly declared in object `x`, such as `x.g`, to be mapped into a data group.

This restriction in the maps clause ensures soundness and modularity by requiring the following data group relationships. First, in the type of `x`, the declaration of `y` must map `y.z` into some data group *G*; thus `y` must be a pivot in `x` and field `x.y.z` would then be a member of data group `x.G`. Next, `x.G` would be mapped into data group *F*; thus `x` must be a pivot field in the receiver `this` and `x.y.z` would be a member of *F* as required. Also, `x.y.z` cannot appear in the represents clause for *F* (Figure 3.10); instead `x.G` must be used. Therefore, `x.y.z` cannot be accessed by *F* unless `x` and `x.y` are pivots. Our technique can then prevent these pivot objects from having aliases that would allow unexpected changes to the value of `x.y.z`; more specifically, only methods of the receiver are allowed to modify pivot object `x`, and only methods of `x` are allowed to modify pivot object `x.y` (Section 3.5). Thus, as required for soundness, unrelated methods are not allowed to change the value of concrete fields in *F*'s data group and, in particular, field `x.y.z`.

3.4.1.2 Additional benefits and a comparison with the Law of Demeter

We also believe that our syntactic restrictions should be followed in general because they have significant advantages for programmers, designers, and verifiers. That is, they seem to result in a better class design since the specification and design will have to follow a discipline similar to the Law of Demeter [LH89]. Obeying the Law of Demeter can result in less coupling between methods, better information hiding, easier method reuse, easier correctness proofs, and many other related benefits [LH89].

The Law of Demeter is a programming style rule that restricts the method calls allowed. Specifically, it only allows calls on the objects in a method's immediate context, such as, static-calls, self-calls, and object-calls on formal parameters and fields directly declared in the receiver; object-calls are also allowed on newly created objects (usually referenced by a local variable) since such objects are not part of a different context. Also, expressions like `x.y` and `x.m1().m2()` are not allowed in program statements and expressions. However, in practice, a few exceptions are allowed [LH89], for reasons such as improved efficiency, but the main principle is to limit the number of objects, types, and methods the programmer needs to know about.

Similarly, our technique limits the specification of method behavior and its implementation to variables in the immediate context. For example, if a concrete field x is a reference type directly declared in the receiver, then a represents clause can only directly access model fields in `x`, e.g. `x.G` is allowed if *G* is a model field; but concrete fields of object `x`, like `x.y`, and fields indirectly declared in `x`, such as `x.y.z`, cannot appear in the represents clause (Figure 3.10). Also, as in the Law of Demeter, a method call like `x.m()` is allowed, but `x.y.m()` is not (Figure 3.10). Thus, our technique

can easily determine from the represents clause when x needs to be a pivot; we then depend on the protection of pivots by the type of x to make expressions like x.*G* and x.m() safe (Section 3.5).

Our technique also restricts program statements, e.g., it does not allow direct assignment to fields of objects other than the receiver (subsections 1.6.6 and 2.4.3). Also, methods cannot make a call like x.y.m() that modifies x.y (Section 3.5). However, our technique does not attempt to enforce the restrictions of the Law of Demeter[17]. Nonetheless, only a few exceptions to this law would be allowed if our tool were to enforce the assumptions needed to allow concrete data refinement (subsection 1.6.6 and Section 2.7). In any case, we do not allow those violations that would cause our reasoning technique to be non-modular or unsound[18].

3.4.2 Determining Data Group Memberships

In this subsection, we formalize our technique for ensuring that a model field is allowed to change when the concrete fields that determine its abstract value change; this is necessary to correctly specify and control side-effects. To accomplish this, as required by our next rule, a model field and the concrete fields it depends on must be members of the same data groups. This rule also ensures that pivot fields accessed in a represents clause have a maps clause, as required by the Pivot Declaration Rule.

Model Field Data Group Rule. Let *V* be a concrete instance field directly or indirectly declared in *C* or in a superclass of *C*. Let *F* be an instance model field directly declared in *C* or in a supertype of *C*. If *V* is accessed on the right-hand side of *F*'s represents clause, then either (i) *V* must be a member of data group *F* or (ii) *V* must be a member of at least one data group and *F* must be a member of all such data groups containing *V*.

A model field, its represents clause, and the fields accessed can be declared in the same or in different types, i.e., in either a supertype or a subtype. Therefore, this rule does not specify where *V*, *F*, or *F*'s represents clause has been declared. However, the various combinations are handled by providing two different ways to satisfy the rule. That is, the alternatives, (i) and (ii), are needed because, in some situations, only one of them is possible. For example, if *V* is declared in the superclass and *F* in the subclass, as in Figure 3.11, then *V* cannot be added to data group *F* (since *F* is not in scope in the superclass where *V* is declared). However, *F* can be added to superclass data groups containing *V*. Thus alternative (ii) is included because we want to allow this situation. That is, *F* can be

17. Strict enforcement of the Law of Demeter is undecidable, but there is a weaker version that is statically checkable [LH89].

18. The reasons for these restrictions also point out the value of Smalltalk's scope rules, i.e., only parameters, fields of the receiver, and global variables are in scope; however, like our technique, a specification language would have to allow model fields of all objects to be visible so clients can reason about method behavior.

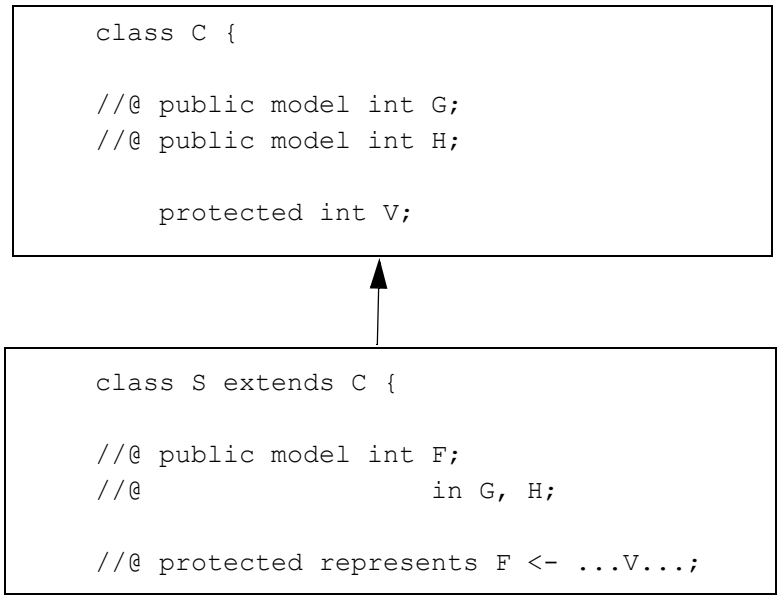

Figure 3.11: Diagram showing the required relationships between superclass and subclass data groups and fields, as required in the Model Field Data Group Rule; this rule requires that F be a member of data groups G and H because F accesses V and V is a member of both G and H.

---

added to the data groups containing *V*, so, as required for soundness, the subclass model field *F* is allowed to change whenever a related superclass field *V* can change.

One of the primary purposes of this rule is make sure that overriding subclass methods have permission to modify a subclass model field *F* whenever assignment to a concrete superclass field *V* could also cause *F* to change. For example, in Figure 3.11, superclass methods that have permission to modify *G* (and implicitly *V*) also have permission to modify *F*. Therefore, overriding methods in subclass *S* can modify *F* (and its members) whenever *G* or *H* are assignable. In fact, methods have additional side-effects if they have permission to modify *G* or *H*; thus these methods have to be overridden and the changes to *F* must be specified for the overriding methods (see subsection 2.2.2). These changes to *F* have to be specified because such methods can assign to *V* which could change the value of *F* (and perhaps require that other of *F*'s dependees be changed), and clients need to know this behavior with respect to *F*.

The Model Field Data Group Rule also has to handle the case where instance field *V* is declared in a subclass. For example, an appropriate represents clause cannot always be declared in interfaces and abstract classes, e.g., when the representation has not yet been implemented; thus a model field can sometimes be declared in a supertype while its represents clause and concrete representation have to be declared in the subtype, as illustrated in Figure 3.12. In such cases, the model field *F* cannot be added to the data groups containing *V*, so *V* must be added to data group *F* (i.e., alternative (i) is the only way to satisfy the rule).

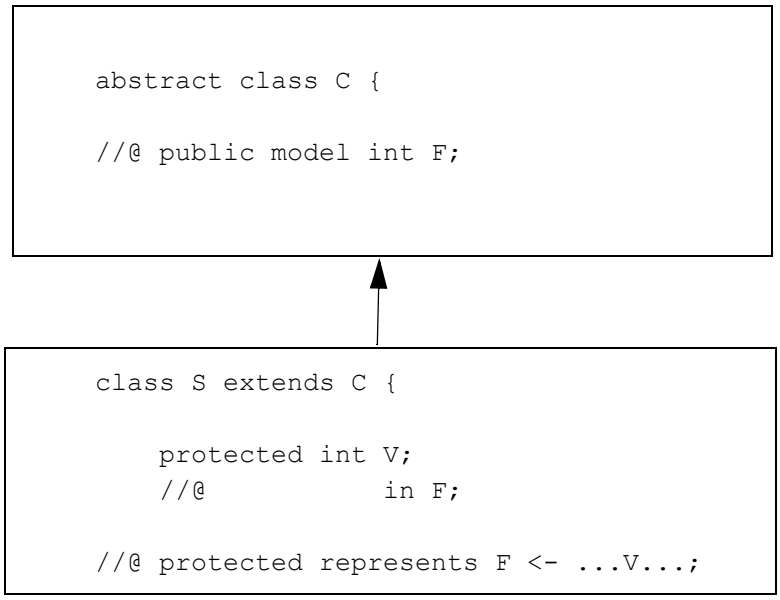

Figure 3.12: Diagram showing the required relationships between superclass and subclass data groups and fields, as required by the Model Field Data Group Rule, that is, V must be a member of data group F.

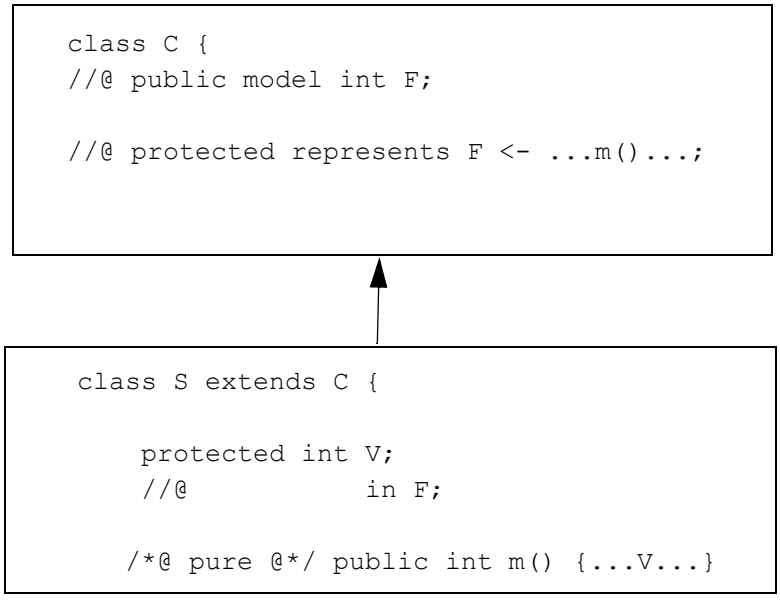

Figure 3.13: Diagram showing the required relationships between superclass and subclass data groups and fields, as required in the Model Field Data Group Rule; this rule requires that V be a member of data group F because F indirectly accesses V.

Notice, also, that a represents clause declared in the superclass can access subclass fields if it calls a method that is overridden, i.e., when the overriding method accesses subclass fields (as in Figure 3.13). However, a superclass model field cannot be added to subclass data groups, so alternative (ii) is not possible. On the other hand, subclass fields can be added to any data group visible

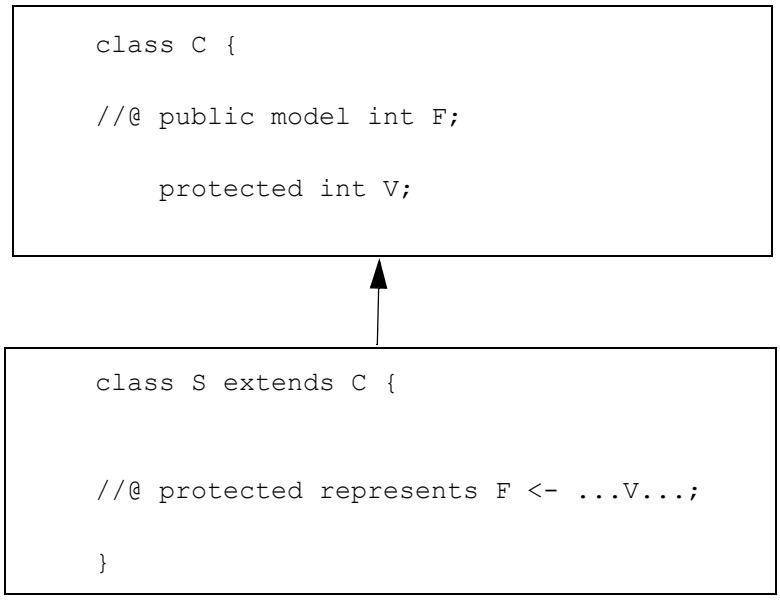

Figure 3.14: Diagram showing the required relationships between superclass and subclass data groups and fields; that is, V must already be a member of data group F because F indirectly accesses superclass field V.

---

to the subclass[19], so alternative (i) can and must be followed whenever subclass fields are accessed by a superclass model field, as shown in Figures 3.12 and 3.13.

Furthermore, when a model field *F* is declared in the superclass and its represents clause is defined in the subclass, then the above rule does not allow the represents clause to access superclass fields that do not already satisfy the above rule. In such cases, the superclass is required to know or anticipate *F*'s and *V*'s potential data group memberships without knowing the representation for *F*, as illustrated in Figure 3.14. However, because our technique assumes that superclasses are correct with respect to their specifications, this situation should not occur very often in practice. In addition, if *V* is not a member of data group *F* in the superclass, then this could mean that there is a more significant problem in the relation between the superclass and subclass definitions, such as the need to refactor the classes involved to make the superclass more reusable by subclasses.

Figures 3.15 and 3.16 provide more examples of the application of the above rule. That is, in Figure 3.15, the in clause in the declaration of `secondSum` is required based on its represents clause given in Figure 3.16; more specifically, `secondSum` needs to be added to the superclass data group `secondValue` because `secondValue` contains the superclass fields accessed by `secondSum`'s represents clause, i.e., `_second.value` and `_second._val`. Furthermore, `secondSum` has to be added to a public data group because a more visible public field like `secondSum` cannot be added to a less visible protected data group.

19.Model fields are normally public, so they would typically be visible to subclasses.

```
public class CellTriple extends CellPair {

   /*@ public model IntCell thirdCell;
     @                    in secondValue;     @*/
   /*@ public model int thirdValue;
     @                   in secondValue;      @*/

   /*@ public model int firstSum;
     @                   in firstCell, thirdValue;       @*/
   /*@ public model int secondSum;
     @                   in secondValue, thirdValue;     @*/

   //@ public invariant thirdCell != null;
   //@ public invariant thirdValue == thirdCell.value;
   //@ public invariant firstSum == firstCell.value + thirdCell.value;
   //@ public invariant secondSum == secondValue + thirdValue;
   //@ public invariant secondSum == secondCell.value + thirdCell.value;

  /*@  public normal_behavior
    @    requires cellOne != null && cellTwo != null;
    @    assignable firstCell, secondCell, thirdCell;
    @    ensures firstCell.value == cellOne.value
    @        && secondCell.value == cellTwo.value
    @        && thirdCell.value == v3;               @*/
  public CellTriple(IntCell cellOne, IntCell cellTwo, int v3);

  /*@  public normal_behavior
    @    assignable \nothing;
    @    ensures \result.value == thirdCell.value;  @*/
    public IntCell third();

  /*@  public normal_behavior
    @    assignable \nothing;
    @    ensures \result == thirdCell.value;  @*/
    public int value3();

  /*@  public normal_behavior
    @    requires newCell != null;
    @    assignable thirdCell;
    @    ensures thirdCell.value == newCell.value;  @*/
    public void setThird(IntCell newCell);
}
```

Figure 3.15: The public specification of `CellTriple` from file `CellTriple.jml-refined`.

```
//@ refine "CellTriple.jml-refined";

public class CellTriple extends CellPair {

    protected IntCell _third;
    /*@               in thirdCell;
      @               maps _third.value \into thirdValue;
      @*/
    //@ protected represents thirdCell <- _third ;
    //@ protected represents thirdValue <- _third.value;

    //@ protected represents firstSum <- _first.value + thirdValue;
    //@ protected represents secondSum <- _second.value + thirdCell.value;
}
```

Figure 3.16: The protected specification of `CellTriple` from file `CellTriple.jml`.

However, in most cases, when this rule is applied, the model field, its represents clause, and its concrete representation are all declared in the same class. Figures 3.1 and 3.2 give examples of the application of this rule when this is the case. That is, the first in clause, in Figure 3.2, satisfies the rule by adding `_first` to data group `firstCell` as required by the represents clause for `firstCell`. Similarly, the maps clause in Figure 3.2 satisfies this rule based on the represents clause for `secondValue`, i.e., the concrete fields in data group `_second.value` (i.e., `_second._val`) have been added to `secondValue`'s data group.

Notice, however, that both alternatives are possible when the model field and the fields it accesses are declared in the same type. Nonetheless, it is simple and usually preferable to add the fields accessed by a model field into that model field's data group, as was done in Figures 3.1 and 3.2, unless the specifier does not want to allow methods to modify all fields that determine the value of *F*; Figures 3.17 and 3.18 provide an example. That is, based on the public specification in Figure 3.17, it would probably be surprising to a client if `x` could also change when method `setY` is called. However, this would be the case if `x` or `_x` were added to data group `sum`. Instead, as in the public specification of Figure 3.17, the above rule can be satisfied for `sum`'s represents clause by adding `sum` to groups `x` and `y`. Therefore, data group `sum` does not contain any concrete variables, so listing `sum` in an assignable clause does not allow `_x` or `_y` to be modified. On the other hand, `sum` is allowed to change whenever either `x` or `y` change, as required for soundness.

3.4.2.1 Summary

Deciding how to satisfy the Model Field Data Group Rule boils down to looking at the location of the declaration of model field *F* relative to *V*'s declaration. That is, model field *F* can be declared in a

```
public class XYPoint {

    //@ public model int x;
    //@ public model int y;

    //@ public model int sum;
    //@                       in x, y;

    //@ public represents sum <- x + y;

  /*@  public normal_behavior
    @     assignable x, sum;
    @     ensures x == newValue;  @*/
    public void setX(int newValue);

  /*@  public normal_behavior
    @     assignable y, sum;
    @     ensures y == newValue;  @*/
    public void setY(int newValue);

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result == sum;  @*/
    public int getSum();

}
```

Figure 3.17: The protected specification for class `XYPoint` from file `XYPoint.jml`.

supertype, the same type, or a subtype relative to *V*, and this determines how the rule can be satisfied. For example, when *F* is declared in a supertype, then *V* has to be a member of data group *F* (as in Figure 3.12). When *F* is declared in a subtype, then *F* has to be a member of all supertype data groups containing *V* (as in Figure 3.11). When *F* is declared in the same type, then both alternatives are available for each variable accessed by *F*; however, if the specifier wants to limit the number of assignable concrete fields when *F* is allowed to change, then the fields added to data group *F* must be limited accordingly (as in Figure 3.17). The location of the represents clause also comes into play, but only when *F* and *V* are declared in a supertype and the represents clause is in a subtype; this situation is only allowed if the data group relationships, declared in a supertype, already satisfy the rule (as in Figure 3.14).

```
//@ refine "XYPoint.jml-refined";

public class XYPoint {

    protected int _x;
    //@                in x;

    protected IntCell _y;
    //@                    in y;
    //@                    maps _y.value \into y;

    //@ protected represents x <- _x;
    //@ protected represents y <- _y.value;

}
```

Figure 3.18: The protected specification for class `XYPoint` from file `XYPoint.jml`.

The Model Field Data Group Rule also makes sure that indirectly declared fields referenced in a represents clause have a maps clause, as required by the Pivot Declaration Rule, since both alternatives require that *V* be a member of a data group. However, if the above rule cannot be satisfied, then unsafe aliasing could result. For example, the Model Field Data Group Rule cannot be satisfied for all fields accessed in the represents clause of `firstSum` shown in Figure 3.16 since `_first.value` is not a member of a data group in the superclass; this could lead to aliasing related problems because `_first` is not a pivot field in the superclass (it would be if `_first.value` were a member of a data group). Therefore, it is an error if a field is not a member of a data group and it is accessed by a subclass model field.

3.4.3 Data groups in the Assignable Clause

The above Model Field Data Group Rule does not ensure that a model field is always allowed to change whenever any one of its dependees can change. For example, two different model fields could depend on the same variable *V*, but this rule does not require that both model fields be assignable whenever one of them could change. However, a change in *V* could change both model fields, so the specification needs to allow both model fields to change. Therefore, we need to include another rule to ensure that all data groups containing variable *V* are assignable whenever any one these groups is listed in an assignable clause.

Assignable Data Group Rule. Let *V* be a concrete instance field directly or indirectly declared in *C* or in a superclass of *C*. Let *F* be an instance model field directly declared in *C* or in a supertype of *C*. If *V* is a member of data group *F* and *V* is assignable in an instance method *M*, then *F* must also be assignable in *M*.

This rule specifies how to check assignable clauses so clients will not be surprised by possible side-effects. That is, this rule makes sure that a method's assignable clause lists all model fields that could change due to assignments during method execution. In this rule, *V* is a concrete field so, in the context of the receiver, *V* will have the form `x` or `x.`*G* with `x` a directly declared concrete instance field and *G* a model field (these are the fields allowed by the syntax of the in and maps clauses).

Therefore, this rule does not apply to model fields. That is, when a model field *F* is listed in an assignable clause, then this rule does not require that groups containing *F* be listed in that assignable clause, unless *F* contains a concrete field; this is important to the way we want to specify side-effects. For example, in Figure 3.17, if the specifier would have inserted `_x` into data group `sum`, then this rule would have required that the specifier also list `x` in the assignable clause of method `setY`. Specifically, concrete field `_x` would have been a member of both `sum` and `x`, so listing `sum` would have required that `x` also be listed. Therefore, this rule would have required a specification that informs clients that `x` could be changed whenever `setY` is called. Instead, `sum` was added to data groups `x` and `y` and `sum` did not contain concrete fields; thus `setY` did not have to list `x` in its assignable clause and `x` cannot be modified.

This rule also limits the data group memberships that can be declared in subclasses. That is, since subclasses can add fields to a superclass data group, this rule has to be applied again for each subclass; thus subclasses cannot add fields that would cause the rule to no longer hold for a superclass method specification. For example, in Figure 3.19, a new concrete field `_sum` is added in the subclass `CacheSum`, so the value of `sum` does not have to be recomputed each time `getSum` is called. Note, however, that `_sum` was not added to data group `sum` because, based on the Assignable Data Group Rule, this would have required that `y` be listed in the assignable clause for `setX` and that `x` be listed in the assignable clause for `setY`. Since this is not the case in the superclass specification, `_sum` must be added to data groups `x` and `y` (just like `sum` in superclass `XYPoint`). That is, `sum` must not contain any concrete fields, otherwise all data groups containing `sum` will have to be listed in an assignable clause whenever any one of them is listed.

Also, this rule is applied locally and modularly in the current type. That is, we only have to be concerned with fields of the receiver in this rule because modification of argument objects must be done through object-calls (see subsection 2.4.3), so methods in the type of an argument object will have to obey this rule; thus the assignable data groups of parameters listed in the assignable clause can and will be derived from the assignable clauses of called methods. Therefore, the Assignable

```
public class CacheSum extends XYPoint {

    protected int _sum;
    //@                in x, y;

    //@ protected invariant sum == _sum;

}
```

Figure 3.19: The protected specification for class `CacheSum` from file `CacheSum.jml`.

Data Group Rule will be satisfied for data groups of parameters based on the specification of the called method.

Also, this rule uses the data group relationships that were required by the Model Field Data Group Rule, i.e., these data group relationships between fields are used in the enforcement of the Assignable Data Group Rule. Therefore, the syntax of the in and maps clauses[20] aids in the enforcement of this rule since these clauses must be part of the declaration of the field being added to a data group (see syntax of the in and maps clauses in Chapter 4 and in our examples); that is, the JML checker can retrieve the associated data group memberships as properties of each field declaration.

3.4.4 Summary

The two rules given in this section provide a principled approach to specifying data group and dependency relationships so side-effects will be properly specified. Thus a primary goal of the rules is to provide a technique for statically checking that side-effects have been properly specified, and thereby prevent some common specification errors.

The Model Field Data Group Rule makes sure that the specification allows a model field to change whenever any of the concrete fields it depends on change. This rule also ensures that pivot fields will have a maps clause, as required by the Pivot Declaration Rule, because any indirectly declared fields accessed in a represents clause must be mapped into a data group, i.e., these assignable concrete fields must be members of a data group.

The Assignable Data Group Rule extends this beyond a single model field by requiring that an assignable clause list all model fields that could change, i.e., model fields with overlapping dependees must all be listed when any one of these model fields is allowed to change. Therefore, these two rules taken together require that the specification inform clients whenever a public model field could change; a side benefit is that specifiers will also know that the specification needs to say how these model fields will change.

20.This syntax is adapted from Leino *et al.*'s paper [LPHZ02].

Notice also that the invariant in `CacheSum` of Figure 3.19 requires that `_sum` be assignable any time `sum` is allowed to change, i.e., the invariant says they must have the same value. As in the Model Field Data Group Rule, there are two ways this can be accomplished, either add `_sum` to `sum`'s data group or add `_sum` to all groups containing `sum` (as explained above, only the second option is compatible with the superclass specification). This example also illustrates that there is a strong similarity in the way fields accessed in the type invariant need to be handled compared with fields accessed by a represents clause.

However, there are also a few significant differences. For example, if our rule, when applied to an invariant, were to require that most fields in the type be members of one large data group, then fine grained control of side-effects may be difficult or impossible. Therefore, the rule needs to focus on grouping the fields that are related. For example, `x+y>0` relates `x` and `y`, but `x>0&&y>0` does not since the constraints are independent in the second expression (i.e., the value of one does not constrain the other). Therefore, all fields accessed by the invariant do not need to be members of the same data group. So to make the rule more practical, each subexpression of the invariant should probably to be handled separately rather than applying the rule to the invariant as a whole. Furthermore, it may be better to allow the specifier to indicate which fields need to be assignable rather than imposing any additional data group membership requirements based on an analysis of expressions in the invariant.

Also, the invariant will frequently need to be specified in the public specification so clients know about it, whereas the represents clause is usually specified in the protected or private specification where concrete fields are visible. Therefore, it would be problematic to require the same syntactic restrictions for both invariant and represents clauses. We, therefore, leave as future work the task of providing a rule and restricted syntax, analogous to the Model Field Data Group Rule and represents clause syntax, needed for statically determining the required data group relationships based on invariant clauses. Nonetheless, the Pivot Declaration Rule requires that indirectly declared fields accessed by the invariant be pivots; we did, however, need to prevent the fields of model objects from being accessed in predicate clauses in general, and invariant clauses in particular (subsection 3.3.4). We believe these restrictions could be weakened, but also leave that as future work.

Another alternative would be try to specify a protected or private invariant that is equivalent to the public invariant; if this is possible, then expressions could be restricted to the syntax of Figure 3.10 and the accessed and related fields of an invariant would be computable. In addition, the JML checker may also be able to detect situations where the invariant may not be implementable, such as when the invariant constrains and relates a set of concrete fields, but there is a method that does not permit all of these fields to be assigned.

Finally, we also found that there were a few situations that were prohibited by our rules. For example, a subclass represents clause for a superclass model field is not allowed to access superclass fields unless the superclass already satisfies the Model Field Data Group Rule for that represents clause. Similarly, a subclass is not allowed to add subclass fields to data groups that would cause a

violation of the Assignable Data Group Rule for superclass specifications. As explained above, these cases cannot be allowed because they could cause our technique to be unsound.

3.5 Protecting Pivot Objects

As illustrated in Section 1.5 and similarly in Figure 3.4, clients cannot reason soundly about programs using public specifications unless the value of the lower-level abstraction (which is hidden from clients) remains synchronized with its higher-level value (which is visible to clients). Therefore, our technique must prevent unexpected changes to the abstract value of model fields through the methods of unrelated objects or classes. Furthermore, clients must not be allowed to inadvertently invalidate the invariant of a type. Therefore, our primary goal is to prevent unexpected changes to the state of a concrete (pivot) object that determines the value of a model field.

The previous section described why the concept of a pivot field is needed and how they are declared in JML. In this section we explain the rules used by our technique to prevent unsound aliasing of pivot objects and how our technique protects the state of these pivot objects when they are aliased.

3.5.1 Owner Variables and Side-effects

The concept of owner variables is used by our technique to restrict method calls that change the state of an existing object. In particular, only an owner variable is allowed to initiate changes to the state of the object it references.

An *owner variable* of an object is the first (in time) variable to contain a reference to that object. Therefore, a variable becomes the owner of an object when it occurs on the left-hand side of an assignment and the right-hand side is a new object constructor call. For example, `_myCell` is an owner variable during the execution of method `increment` in Figure 3.22 because of the assignment statement in that method. Similarly, `_myCell` is an owner variable during the execution of the constructor in Figure 3.21 because the initializer in the declaration of `_myCell` makes it the first variable to contain the reference to a newly created object.

The main principle in our technique is that we want to make sure that all changes to the state of an object are known in the context of its owner variable so clients and verifiers can reason about the state of that object. Also, clients often need to modify the abstract value of an object and this abstract value may depend on the state of a pivot object. Therefore, our technique must also ensure that every pivot field is always an owner variable so any needed changes to a pivot object will be allowed (see also subsection 3.5.2).

Recall that modifying the state of an object, other than the receiver, requires an object-call because direct assignment to fields of other objects is not allowed (see assumptions in subsection 3.5.7). Therefore, when an object-call modifies the state of its receiver and the receiver is a variable, then this receiver parameter must be an owner variable; this restriction ensures that an owner variable initiates

```
public class CellHolder {

   //@ public model IntCell theCell;

  /*@  public normal_behavior
    @     requires cell != null;
    @     assignable theCell;
    @     ensures theCell.value == cell.value;   @*/
    public CellHolder( IntCell cell );

  /*@  public normal_behavior
    @     assignable theCell;
    @     ensures theCell.value == \old(theCell.value) + 1;   @*/
    public void increment();

  /*@  public normal_behavior
    @     requires cell != null;
    @     assignable cell.value;
    @     ensures cell.value == \old(cell.value) + 1;   @*/
    public void add1( IntCell cell );

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result.value == theCell.value;  @*/
    public IntCell getCell();
}
```

Figure 3.20: Public specification of `CellHolder` from file `CellHolder.jml-refined`.

all changes to the object it references. For example, the side-effects in the second statement of the constructor shown in Figure 3.21 are allowed because `_myCell` is an owner variable in this context.

However, in addition to the receiver, we also want to allow methods to modify objects passed as explicit arguments. To do this, we have to allow formal parameters to become *temporary owner variables* during a method call. Therefore, to make sure such calls do not modify an object owned by a variable in a different context, we have to restrict method calls that modify the state of all argument objects. Our next rule ensures that, during a method call, all changes to an object are initiated by an owner so clients and verifiers can reason about the state of that object in the context of its owner variable (this also assumes there are specifications of the behavior and side-effects of the called method).

```
public class CellHolder {
    protected IntCell _myCell = new IntCell(0);
   //@                in theCell;
   //@                maps _myCell.value \into theCell;
                   // (0) _myCell is a pivot field

    public CellHolder( IntCell cell ) {
        _myCell.setValue(cell.getValue());
                   // (1) OK because _myCell is a pivot field
    }
    public void increment() {
        add1(_myCell);
                   // (2) allowed because _myCell is a pivot field
    }
    public void add1( IntCell cell ) {
        cell.setValue(cell.getValue()+1);
                   // (3) OK because cell is an owner variable
    }
    public IntCell getCell() {
        return _myCell;
    }
}
```

Figure 3.21: Implementation of `CellHolder` when `_myCell` is a pivot field.

Owner Variable Rule. When a method is allowed to assign to fields of the receiver or formal parameter object, then the corresponding actual parameter must be an owner variable, `null`, or a new object constructor call.

This rule says that when an object-call is allowed to change the state of the receiver object, then the receiver in that object-call must be an owner variable or a new object constructor call (`null` is not a valid receiver in Java). Therefore, the receiver cannot be a method call expression since the owner would not be known in this context and thus cannot know about or initiate these changes. For example, `getCell().setValue(1)` is not allowed by this rule because the receiver is not an owner variable or new object constructor call.

This rule also disallows the call of `add1` in method `increment` of Figure 3.21 when `_myCell` is not a pivot field. However, to allow that call of `add1`, `increment` could be implemented as in Figure 3.22. Notice that, unless a field is a pivot, ownership does not continue after execution of the method where the field was first assigned a reference to a new object; this allows the checking to be done statically and modularly as described in subsection 3.5.5. Furthermore, if `_myCell` is not a

```
public class CellHolder {
    protected IntCell _myCell;
   //@                in theCell;
                   //  (0) _myCell is not a pivot field

    public CellHolder( IntCell cell ) {
        _myCell = cell;
                   // (1) not allowed if _myCell is a pivot field
    }
    public void increment() {
        _myCell = new IntCell(_myCell.getValue());
        add1(_myCell);
                   // (2) OK because _myCell is a temporary owner
    }
    public void add1( IntCell cell ) {
        cell.setValue(cell.getValue()+1);
                   // (3) OK because cell is an owner variable
    }
    public IntCell getCell() {
        return _myCell;
    }
}
```

Figure 3.22: Implementation of `CellHolder` when `_myCell` is not a pivot field.

pivot, then `CellHolder` cannot care about or reference the state of object `_myCell` in its specification since that object may be owned by and thus modified through some other variable.

The above rule ensures that an owner variable has initiated and knows about any side-effects to the object it references. The rule given in the next subsection explains how we make sure that pivot fields and formal parameters are always able to initiate changes to the objects they reference.

3.5.2 Owner Variables and Pivot Fields

The purpose of our next rule is to make sure that a pivot field is always the owner of the object it references and that no two pivot fields are static aliases for the same object. In other words, a pivot field must always contain the reference to a newly created object and different pivot fields must not reference the same pivot object.

However, pivot objects can safely be aliased temporarily during method calls made by the owner. Such temporary aliasing is safe unless the call could result in unexpected side-effects or certain kinds of static aliasing of a pivot object. Nonetheless, in most cases, pivot objects can safely have dynamic aliases via parameters, because these aliases are temporary with a specific lifetime that is known to and can be controlled by an owner variable.

Since no two pivot fields can be allowed to reference the same object at run-time, we need to look at assignment statements because, as described at the beginning of this chapter, static aliases are created through assignment to fields. For example, in Figure 3.3, the assignments to pivot field `_second` in the constructor for `CellPair` as well as in method `setSecond` could result in a static alias to `_second`. This is illustrated in Figure 3.4 where the constructor that initializes local variable `pair2` creates a static alias to pivot field `_second` because external object `c3`, one of its actual parameters, is assigned to pivot field `_second`. To avoid this situation, our first rule restricts what can be assigned to a pivot field to ensure that the reference will be assigned to at most one pivot field or formal parameter in the system; this rule also ensures that pivot fields and formal parameters are owner variables when the pivot object is mutable.

Pivot Assignment Rule. When a pivot field or formal parameter is the left operand (target) of an assignment statement, then the right operand must be `null` or a new object constructor call, unless the type of the target variable is immutable or a primitive type.

This rule ensures that, after an assignment, a pivot field is the first variable to contain a reference to a pivot object whenever that object is mutable; therefore, a pivot field is always an owner variable. Furthermore, this restriction prevents any two pivot fields from being statically aliased since only one pivot field can be the first to contain the reference. However, this rule does not prevent any other aliasing in the system, but rather our technique protects the state of pivot objects by disallowing all other aliases (i.e., non-owner variables) from modifying the state of pivot objects (the Owner Variable Rule in subsection 3.5.1 accomplishes this restriction on side-effects).

A correct implementation of `CellPair` is shown in Figure 3.23. This implementation does not have the aliasing problems shown in the constructor and `setSecond` of Figure 3.3. That is, both assignment statements to `_second` now create a new internal pivot object rather than creating a static alias to the parameter objects `cellTwo` and `newCell`; to be safe, our assumption is that an argument object may already be owned by another variable in the system.

Furthermore, in class `CellPairClient` of Figure 3.4, local variables `c1` and `c2` are not owner variables, so the object-calls (to `setValue`) that modify these objects would not be allowed. However, `c3` is an owner variable so the object-call that modifies the state of `c3` would be allowed; also, object `c3` cannot and will not be referenced by any pivot field in the system.

Figure 3.21 provides a valid implementation of class `CellHolder` when `_myCell` is a pivot field. However, when `_myCell` is not a pivot field, that same class would have to be implemented differently, as shown in Figure 3.22. For example, the assignment in the constructor in Figure 3.22 would not be allowed if `_myCell` is a pivot because this could create a static alias with another pivot field or owner variable in the system. Also, as explained earlier, if `_myCell` is not a pivot, then the implementation of method `increment` in Figure 3.21 would not be allowed because the call to `add1` could modify the state of an object referenced by a non-pivot field; conversely, `increment` could be

```
//@ refines "CellPair.jml";

public class CellPair {

    protected IntCell _first;
    protected IntCell _second;
    protected int _val2;

    public CellPair(IntCell cellOne, IntCell cellTwo) {
        _first = cellOne;
        _val2 = cellTwo.getValue();
        _second = new IntCell(_val2);
    }
    public /*@ pure @*/ IntCell first() {
        return _first;
    }
    public /*@ pure @*/ IntCell second() {
        return _second;
    }
    public /*@ pure @*/ int value1() {
        return  _first.getValue();
    }
    public /*@ pure @*/ int value2() {
        return  _val2;
    }
    public void setFirst(IntCell newCell) {
        _first = newCell;
    }
    public void setSecond(IntCell newCell) {
        _val2 = newCell.getValue();
        _second = new IntCell(_val2);
    }
}
```

Figure 3.23: A correct implementation of `CellPair` in file `CellPair.java`.

implemented as shown in Figure 3.22 because the assignment statement makes `_myCell` a temporary owner variable.

The above rule is similar to one of the restrictions given in Leino *et al.*'s paper [LPHZ02], except that we do not restrict the aliasing of objects in immutable types. This exception is allowed because the abstract value of objects in such types cannot be changed. Subsection 3.5.4 more fully describes our

```
public class IntCell {
   //@ public model int value;      // model variable

   ...

  /*@  public normal_behavior
    @     requires c != null;
    @     assignable value;
    @     ensures value == c.value;  @*/
   public void setFrom(IntCell c);
}
```

Figure 3.24: A fragment of the public specification of `IntCell` from file `IntCell.jml-refined`.

```
//@ refines "IntCell.jml-refined";

public class IntCell {

  protected int _val;
  //@              in value;

  //@ protected represents value <- _val;

   ...
}
```

Figure 3.25: A fragment of `IntCell`'s protected specification from file `IntCell.jml`.

definition of an immutable type and why objects in an immutable type can safely be aliased among pivot fields.

3.5.3 Aliasing of Actual Parameters

The above two rules ensure that a pivot object can only be changed through its owner variable (a single pivot field) and that pivot fields always own the object they reference. However, there is still another aliasing problem that our technique needs to handle. Consider for example, the specification and implementation of method `setFrom` given in Figures 3.24-3.26. In this example, there is clearly the possibility of an unexpected side-effect that changes the behavior of method `setFrom`. That is, in the implementation of `setFrom` in Figure 3.26, if the receiver and the formal parameter are aliases,

```
//@ refines "IntCell.jml";

public class IntCell {

  protected int _val;

   ...

  public void setFrom(IntCell c) {
      _val = 0;
      _val = c.getValue();
  }
}
```

Figure 3.26: A fragment of `IntCell`'s implementation from file `IntCell.java`.

then the value of `this._val` will always be 0. The implementation satisfies the specification, but it is unlikely that this is the behavior intended by the specifier or the implementer.

Furthermore, such aliasing means that there could be assignments to fields of a formal parameter (or the receiver) even though the assignable clause says that that argument object is not changed. This makes reasoning difficult because the verifier always has to consider the special case(s) when formal parameters or the receiver could be aliased. We might have left the problem for the verifier except that this is also a problem for the soundness of our technique. For example, the code of each method of a class is statically analyzed using method specifications and our technique needs to know which fields are assignable. In particular, our technique depends on the assignable clause, so soundness requires that the assignable clause specify all side-effects allowed to fields of the receiver and formal parameters. Furthermore, we do not want changes to the state of a formal parameter object to possibly invalidate the invariant of the receiver. Therefore, unless we prevent such aliasing, assignments to fields of a formal parameter could change the state of the receiver and vice versa. So our next rule prevents objects from being aliased through formal parameters or the receiver when there are side-effects to the same fields of these aliased objects.

Actual Parameter Aliasing Rule. An argument object of a method call cannot have assignable fields if the called method directly or indirectly accesses those assignable fields through a different access path.

This rule only disallows aliasing when the fields that can be changed are also accessed (read) in the called method through an alias. For example, the call of method `add1` in method `increment` of Figure 3.21 does not cause a problem and is allowed by the Actual Parameter Aliasing Rule because

fields of the aliased pivot object are not accessed through the receiver of `add1`. However, the accessible clause would be needed to enforce the above rule modularly. That is, without the accessible clause, the checker would have to analyze the code of all directly and indirectly called methods to determine whether or not an assignable field may be accessed. Since we are leaving the use of the accessible clause as future work[21], we have formalized, in subsection 5.1.2.4, a more conservative rule, i.e., we do not allow aliasing between the receiver (or its pivot fields) and the formal parameter when there are side-effects. For example, the call of method `add1` in method `increment` of Figure 3.21 would not be allowed by the formalization in Chapter 4, but this call could be allowed if we were to enforce the above rule using the accessible clause. Note that this less flexible rule is similar to the owner exclusion restriction in Leino *et al.*'s paper [LPHZ02].

3.5.4 Immutable Types

Our technique must protect the state of pivot objects because, as described previously, pivot objects contain the data values related to the higher-level abstraction or to the type invariant; to accomplish this we have to prevent any representation exposure that would allow a mutable pivot object to have more than one pivot field owner, i.e., static aliasing via more than one pivot field.

However, static aliasing of pivot objects is not unsafe when the abstract value of the pivot object is immutable. Therefore, we would like to allow static aliasing of immutable pivot objects. We say that a *type* is *immutable* if there are no methods in the system that can modify the abstract value of objects in that type. Also, an *object* is *immutable* if it is in an immutable type.

In JML, the modifier pure in the type header says that all of the methods declared in this type are pure, i.e., do not have side-effects. We refer to such types as *pure types*. However, the pure modifier only specifies that the current type has no methods with side-effects, i.e., it does not guarantee that its supertypes and subtypes are pure. Therefore, pure types are not necessarily immutable. Therefore, to be immutable, a type must be pure and all of its supertypes, except the root class `Object`, must also be pure[22]. In addition, an immutable type must not have any non-pure subtypes; thus, to guarantee this, we require that an immutable type be final. However, final is not a valid modifier for interfaces in Java, so only classes can be immutable types.

---

21. The accessible clause would have to be modularly enforced with rules similar to those given in subsection 5.1.7 for enforcing the assignable clause; enforcement of the accessible clause would also have to deal with all possible aliases.

22. The root class (`java.lang.Object`) in the Java class hierarchy is not pure, i.e., it has methods with side-effects that are used for thread synchronization and garbage collection. However, in most applications, a specification rarely, if ever, accesses the state of this root class in a represents clause. Since we are only concerned about unexpected changes to the abstract state of an object, we, therefore, do not have to be concerned about the root class when determining whether a type is immutable. Nonetheless, if, for some unusual reason, the mutable state declared in `java.lang.Object` is accessed in a represents clause of a subclass, then such objects would not be considered immutable.

```
public final /*@ pure @*/ class PureCellPair {

   //@ public model IntCell firstCell;
   //@ public model IntCell secondCell;
   //@ public model int secondValue;

   //@ public invariant firstCell != null && secondCell != null;
   //@ public invariant secondValue == secondCell.value;

  /*@  public normal_behavior
    @     requires cellOne != null && cellTwo != null;
    @     assignable firstCell, secondCell;
    @     ensures firstCell.value == cellOne.value
    @        &&  secondValue == cellTwo.value;
    @*/
    public PureCellPair(IntCell cellOne, IntCell cellTwo);

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result.equals(firstCell);
    @*/
    public /*@ pure @*/ IntCell first();

  /*@  public normal_behavior
    @     assignable \nothing;
    @     ensures \result.value == secondValue;
    @*/
    public /*@ pure @*/ IntCell second();
}
```

Figure 3.27: Public specification of `PureCellPair` from file `PureCellPair.jml-refined`.

In addition, it is not sufficient that a class be final and pure and that all of its superclasses be pure. That is, our technique must also make sure that the abstract value of an object in an immutable type cannot be changed by methods of unrelated types; that is, we have to deal with the same kinds of aliasing problems even when the enclosing class has no methods with side-effects. Thus the same rules needed for mutable types have to be applied to immutable types. For example, Figures 3.27 and 3.28 show the public and protected specifications of `PureCellPair`. Figure 3.29 gives its implementation. This class is the same as the `CellPair` class shown in Figures 3.1-3.3 except that it is final and pure, and the two methods with side-effects have been removed. However, this new class has some of the same problems as `CellPair` because the constructor assigns parameter `cellTwo` to

```
//@ refines "PureCellPair.jml-refined";

public final /*@ pure @*/ class PureCellPair {

    protected IntCell _first;
    //@               in firstCell;

    //@ protected represents firstCell <- _first;

    protected IntCell _second;
    /*@               in secondCell;
      @               maps _second.value \into secondValue;
      @*/

    //@ protected represents secondCell <- _second;

    //@ protected represents secondValue <- _second.value;

    protected int _val2;
    //@           in secondValue, secondCell;

    //@ protected invariant secondValue == _val2;
}
```

Figure 3.28: Protected specification of `PureCellPair` from file `PureCellPair.jml`.

pivot field `_second`, but the constructor has no way of knowing whether or not there is another static alias to this object elsewhere in the system. Thus other owner variables may be able to change the state of object `_second` which could change the abstract value of the type which could also invalidate the invariant shown in Figure 3.28. Consequently, even when a class is pure and final, the specifier must declare the pivot fields with a maps clause (i.e., follow the Pivot Declaration Rule) so mutable pivot objects will not be aliased between what should have been a pivot field and other owner variables. Thus the same rules have to be applied to pivot fields in an immutable type so its abstract value cannot be changed by unrelated methods.

3.5.5 Enforcement of these Rules

Our tool will enforce the above two rules by associating a boolean owner attribute with each variable. Prior to method execution (i.e., analysis of the method body), this attribute is true for all pivot fields and formal parameters and false for all non-pivot fields and local variables. Each assignment statement in the method body changes this attribute, i.e., either makes the attribute of the variable on its left-hand side true or false depending on the expression on the right-hand side. However, if the

```
//@ refines "PureCellPair.jml";

public final /*@ pure @*/ class PureCellPair {

    protected IntCell _first;
    protected IntCell _second;
    protected int _val2;

    public PureCellPair(IntCell cellOne, IntCell cellTwo) {
        _first = cellOne;
        _second = cellTwo;
        _val2 = _second.getValue();
    }
    public /*@ pure @*/ IntCell first() {
        return _first;
    }
    public /*@ pure @*/ IntCell second() {
        return _second;
    }
}
```

Figure 3.29: An incorrect implementation of `PureCellPair` from file Pure`CellPair.java`.

assignment would make the owner attribute of a pivot field false, then it is an error (i.e., a violation of the Pivot Assignment Rule). Self-calls that are permitted to assign to a non-pivot field of the receiver make the owner attribute of that field false. The program analysis for determining the value of this owner attribute would be similar to what Java and JML do when checking whether a variable has been initialized prior to first use.

Also, during the analysis of the method body, we need to check method calls, i.e., enforce the Owner Variable Rule. Therefore, if the called method has permission to modify the state of an argument object, then the corresponding actual parameter must be a new object constructor call or a variable reference with a true owner attribute (or `null` when allowed).

3.5.6 Side-effects to Objects Referenced by a Local Variable

Local variables can also be temporary owners of an object; they just have to be the first variable containing the reference to a newly created object. Note, however, that when a local owner variable goes out of scope, then the object it referenced no longer has an owner and that object cannot be modified later. That is, we do not allow ownership to be transferred to a different variable. However, as future work, we would like to be able to transfer ownership from a local variable to a pivot field or from one pivot field to another when it is safe to do so[23].

Therefore, in our technique, it is possible for an object to have no owner, one owner, or several owner variables. For example, an object may not have an owner variable if its original owner is a local variable or non-pivot field, since an object can only be owned during the lifetime of its owner variable or until another assignment makes that variable a non-owner. In contrast, during a method call, an object may have more than one owner. For example, during the call of `add1` in method `increment` of Figures 3.21 and 3.22, the argument object `cell` will have at least two owners since the call is not allowed unless the actual parameter `_myCell` is an owner variable (in addition to the formal parameter `cell`).

Note that formal parameters always start out as owner variables prior to method execution because the corresponding actual parameters are owner variables or new object constructor calls. However, the Pivot Assignment Rule does not require that a formal parameter remain an owner. That is, an assignment to a formal parameter (although rare in most programs) could change this ownership property just like it does for any other local variable or non-pivot field. Nonetheless, pivot fields must always be owner variables so checking of expressions and statements involving pivots can be done statically and modularly (see subsection 3.5.5).

3.5.7 Assumptions

In addition to the two rules given in this section, we restrict changes to internal objects in several other important ways, including visibility restrictions. That is, as described in Chapters 1 and 2, we make a few additional assumptions (restrictions) that are necessary for the soundness of our technique. For example, we do not allow concrete fields to have public visibility; we also require that changes to the state of an object, other than the receiver, be done through object-calls. That is, we only allow direct assignment to fields of the receiver `this` (see subsection 1.6.6 and 2.4.3); thus our technique does not allow a method to make direct assignments to fields of other objects, such as `_second._val` in Figure 3.2, or to a field of a parameter object.

3.6 Discussion

In this section, we first investigate some of the ways that our rules could be made less conservative, i.e., in the first three subsections we show that more expressions could be allowed in represents and predicate clauses. However, the rules given in these subsections have to be applied by hand because they are not handled automatically by our tool. Next, in subsection 3.6.4, we give an overview of why we believe that our technique is sound. Finally, in subsection 3.6.5, we discuss the

23. This would require adding a feature like a \not_owned(E) expression or a linear type system with a destructive read operation that specifies that the object returned from expression E is guaranteed to never be referenced in the future by the original owner variable (or pivot field) in the system. This would allow a new variable to acquire ownership of that object and be the only variable allowed to initiate changes to that object. However, we leave this extension as future work.

contributions and limitations of our alias control technique; we also describe some additional future work.

3.6.1 Fields of Model Objects Revisited

As mentioned in subsection 3.3.4, for soundness, a proof may sometimes be required to ensure that every concrete object *O* is a pivot whenever fields of *O* determine the value of a model field *F.x* and *F.x* is accessed in a predicate clause. Furthermore, if the concrete fields that determine the value of *F.x* cannot be precisely determined (by hand or automatically), then all objects reachable through *accessed*(*F*) must be pivots. Thus if all objects reachable through a model object *F* are pivots, then the state of *F* can safely be accessed in a represents or predicate clause, because all internal objects would be encapsulated. Our next rule allows this.

Reachable object rule. Let *F* be a model field with a reference type. If every object reachable through a field in *accessed*(*F*) is a pivot object, then the state of *F* can be accessed in represents and predicate clauses.

This rule says that the state of a model object *F* can be accessed by the invariant if no object reachable through fields in *accessed*(*F*) can be aliased. Thus this rule allows the second invariant clause in Figure 3.1 because all objects reachable through fields accessed by `secondCell` are pivot objects, i.e., `_second` is a pivot and `IntCell` does not have any internal objects. On the other hand, this rule would not allow `cell.value` in the second invariant clause of Figure 3.9 because `cell` accesses `_pair._first`, a non-pivot object. Similarly, the first invariant clause would not be allowed because `pair.firstCell.value` indirectly accesses `_pair` and a non-pivot, `_pair._first`, is reachable through `_pair`.

However, it is important to note in the rule that if model instance field *F* is declared in an interface, then the above rule cannot be applied because there is no concrete implementation and no represents clause. Therefore, *accessed*(*F*) cannot be calculated until the concrete class that implements that interface is defined and *F*'s represents clause has been declared.

Furthermore, this rule may not be applicable in many situations, e.g., when the type of an internal object is in a different package. That is, only the public specification of types in a different package are visible so it is usually not possible to determine whether all reachable objects are pivots. Therefore, this rule is not usually going to be useful for customizers extending classes from a framework or class library; however, it is presented here for those situations when all the classes referenced in a type are in the same package. Furthermore, our tool will not check for these situations.

3.6.2 Accessing Private Fields

When extensible types are specified in layers of abstraction, as can be done in JML, a model field will normally be public and it will access protected concrete fields. However, as described in subsection 2.6.1.1, class libraries may not want to expose concrete fields declared in a superclass to its subclasses. Thus if the accessed fields are private (and they are not declared spec_protected), then it

may not be possible for a customizer creating a subclass to determine whether or not the required objects are pivots since the private specification of the superclass would not be available.

For example, suppose that Figure 3.2 is the private specification of `CellPair` and that the access modifiers shown there are private rather than protected. In this situation, the customizer would only have the public specification shown in Figure 3.1 from which to work. However, the public specification says nothing about the pivot fields of `CellPair`[24]. Therefore, because model field `firstCell` may have accessed a private object, we cannot allow an expression like `firstCell.value` in subclass predicate clauses, e.g., invariant clauses. However, we can allow such expressions in a subclass predicate clause if that expression has already been used in a superclass predicate clause. For example, based on Figure 3.1, any subclass of `CellPair` can assume that all the required pivots have been declared for expression `secondCell.value` since that would have been required in superclass `CellPair`. Thus `secondCell.value` can be used in a subclass invariant. Our next rule formalizes these requirements and this exception.

Private object access rule. Let *C* be a concrete class. Let *F* be a model field with a reference type directly declared in *C*. If the represents clause for *F* is not visible to a type *T*, then a field of model object *F* cannot be accessed in any of *T*'s predicate clauses unless that field is accessed in one of *C*'s predicate clauses visible to *T*.

In our technique, to allow more possibilities in subclass predicate clauses, the represents and maps clauses will be visible to customizers of a concrete class even though a model field accesses private fields[25]. That is, we assume (and require) that the represents and maps clauses for each field be visible to customizers whenever possible, i.e., appear in the protected specification. However, if the represents clause of model field *F* is not visible in a subclass *S* of *C*, then the verifier of the subclass will have no idea whether the concrete fields accessed by fields of *F* are pivots (since the related maps clauses will also not be visible). In such situations, only expressions involving *F* that already occur in *C*'s predicate clauses can appear in *T*'s predicate clauses.

Notice also that this rule is intentionally worded so it also applies when *C* and *T* are unrelated classes, e.g., when *T* declares an internal object with type *C*. Thus if the represents and maps clauses related to model field *F* are not visible in type *T* (usually the case when *C* and *T* are in different packages), then only fields of *F* that occur in *C*'s predicate clauses can be accessed by *T*'s predicate clauses.

---

24. The protected specification would include accessible and callable clauses for public and protected fields and methods, but this information does not say anything about private pivot fields.

25. Note that abstract classes are not required to have a represents clause. Therefore, to keep things simple, when an abstract class does not have a represents clause for a model field, we assume that it is to be declared by a subclass in the protected specification (otherwise we would have to assume that the represents clause is private, which is more restrictive in the subclasses allowed).

Even though this rule is not enforced by our tool or technique, it is provided here so customizers can use it to permit expressions that would not otherwise be allowed when the superclass has private concrete fields. However, this rule is also useful because it can be applied when the represents clause is visible; that is, references to fields of *F* that occur in *C*'s predicate clauses can always be accessed by *T*'s predicate clauses because the required pivots will have been declared in *C*.

3.6.3 Static Fields

Static fields present an interesting and unique set of problems. This is because static fields are global variables that introduce problems similar to those caused by aliasing (even when they reference an unaliased object). That is, static fields can be accessed and modified by unrelated objects because they are visible to all the unrelated objects of a particular type. Therefore, for soundness, we cannot allow a represents or predicate clause to access static fields unless they are, in effect, constants, i.e., immutable.

A field is *immutable* if, after initialization, there are no methods or types that can assign to that field. For example, a field is immutable, in Java, if it is declared with the final modifier. Similarly, in JML, a constraint clause, such as `\old(x)==x`, prevents assigning a different value (e.g. reference) to `x` after initialization. However, disallowing the modification of a static field is not sufficient for soundness when that field is a pivot; that is, a pivot references an object, and, therefore, that object must also be immutable since its abstract value determines the abstract value of all objects in the type.

In the rules given in Section 3.5, our technique allowed pivot objects in immutable types to be aliased. Similarly, an immutable (e.g. final) static field that references an object in an immutable type could be a pivot without affecting the soundness of our technique. Nonetheless, in this dissertation, we assume that types do not declare static fields (see assumptions in subsection 1.6.6); we leave as future work the rules necessary for handling static fields.

3.6.4 Soundness of Our Technique

Our technique is sound if the abstract value of every object in the system cannot change unless the specification of a called method allows it to change. That is, our technique must not allow the abstract value of any object in the system to change unexpectedly. We accomplish this in two ways. First, our technique hides concrete fields from unrelated classes and objects through visibility control and layers of abstraction. For example, public model fields must be used to hide the underlying implementation, and the behavior of public methods must be specified in terms of these public model fields. Furthermore, concrete fields are not allowed to have public visibility and methods can only assign to fields of the receiver (see assumptions in subsection 1.6.6).

However, when the state of an internal object determines the abstract value of a model field, then such objects must be pivots so they cannot be changed by methods in unrelated types or objects. Therefore, the second way our technique prevents unexpected behavior is by ensuring that all pivot fields in every object are declared and that the pivot objects referenced are not exposed to methods that

could create an alias with another owner variable. The rules given in Sections 3.3 and 3.4 make sure that all pivot fields, accessed by model fields and predicate clauses, have been declared. The rules in Section 3.5 ensure that all changes to pivot objects are initiated by pivot fields so verifiers can reason soundly about the state of pivot objects and thereby ensure that clients can reason soundly from specifications. Chapter 4 provides a proof that these rules are sufficient to guarantee that the abstract value of the objects in the system cannot change unexpectedly.

3.6.5 Contributions

Our rules require that a field have a maps clause whenever it references a mutable object and the state of that object is accessed in an represents or predicate clause. Our technique also uses these maps clauses to specify the pivot fields of a type. An important advantage of using the maps clauses is that implementers can determine the pivot fields at a glance without having to look at all the represents and predicate clauses, since syntactically the maps clause must be part of a field declaration. Furthermore, no additional annotations have to be added to our language (or in most cases to the specification) because, when a field is a pivot, the maps clause is usually required to properly control side-effects. The maps clause is also the easiest way for tools to determine the pivot fields of a type; our tool can also statically check that the pivot fields have been properly declared based on the represents and predicate clauses. That is, our tool can use the accessible and callable clauses (also generated by our tool) for all methods directly or indirectly called to determine which fields of a pivot object are indirectly accessed; this would then be used to ensure these fields have a maps clause.

On the other hand, the specifier could add an indirectly declared field to a data group even though that field is not accessed in a represents or predicate clause and thus may not need to be treated as a pivot. Nonetheless, this does not affect the soundness of our technique and we believe this should be an unusual situation in practice.

The Model Field Data Group Rule and the Assignable Data Group Rule of subsection 3.4.2 provide a principled technique for specifying the dependencies among concrete instance fields when they are related to a model field through its represents clause; also, as explained in that subsection, these dependency relationships are necessary to specify and control side-effects. Other related techniques [LPHZ02, LN02, Mül01, Mül02] require a rule similar to the Model Field Data Group Rule but none of those that we know about allow represents or invariant clauses to relate superclass and subclass fields; this is handled by the subclass model field rule given in subsection 3.4. Furthermore, because these rules require that an object be a pivot whenever its state is accessed in a represents clause, the Pivot Declaration Rule does not have to be applied for represents clauses; it only has to be applied to predicate clauses. These rules also make sure that the rules given in Chapter 2 can be applied and checked.

The enforcement of the assignable clauses may also require that additional fields be declared as pivots. So most, if not all, pivot fields have to be declared to allow the required side-effects. However,

to be sure that all situations requiring a pivot field are properly specified, the Pivot Declaration Rule and the algorithm for calculating the fields accessed in an expression were given; the Pivot Declaration Rule also provides one simple, comprehensive criteria for static checking, namely, that all pivot fields have been declared.

We are not aware of any existing, sound technique [Lei95, LN02, Mül01, Mül02] that allows accesses of model fields or method calls in what would be equivalent to our represents clause; thus our technique is an extension. Furthermore, the existing techniques do not allow mappings from concrete object structures to abstract object structures or the access of the state of abstract objects in the type invariant. Finally, our technique allows relationships between superclass fields and subclass fields in the represents and predicate clauses (e.g., invariants) which we also believe to be new. However, our technique, as described here, is only a partial solution to some of the problems related to model objects in type specifications; we leave as future work a complete (or more complete) solution, and in particular, the relaxing of some of the assumptions and syntactic restrictions of our technique.

An important advantage of our technique for protecting pivot objects is that it only needs the annotations in the maps and assignable clauses that are already required in JML specifications (see subsection 3.5.5). Even though the rest of the JML specification is not used in the enforcement of the rules for protecting internal objects, a more complete specification is needed if one needs to reason about the state of an object without the code. The specification of method behavior is needed by clients, customizers, and verifiers and the represents clauses are needed by customizers and verifiers. Our tool also uses the represents clauses to determine which internal objects need to be protected, i.e., need to be pivots (see Sections 3.3 and 3.4). Examples of other systems that prevent unsound aliasing and side-effects are given in Chapter 7; all of them require additional annotations in the specification language. A major advantage of our technique is that our rules are very simple and easy to enforce compared to many of the other techniques, particularly those that need to enforce the encapsulation of references.

CHAPTER 4: THE PROGRAMMING AND SPECIFICATION LANGUAGES

In this chapter we define the syntax and semantics of the programming and specification languages that will be used in our soundness proof. In Section 4.1, we briefly review the Java Modeling Language (JML) and the additional specification information needed by our technique and provided in the subclassing contract. We next define, in Section 4.2, the syntax of the programming language which, for simplicity, only includes the important features of Java, i.e., the features needed to show that our technique works for a single dispatch object-oriented language. In Section 4.3, we define the syntax of the core specification language; similarly, we only include the features of JML needed to demonstrate the soundness of our technique. The core programming and specification languages will be referred to as Java-C and JML-C respectively. In Section 4.4, we describe the type environment that will be used in our semantic model of dynamic binding. Since our goal is to show the soundness of our technique, we first need an operational semantics; in Section 4.5, we define an operational semantics for Java-C, including a model of dynamic binding, objects, and the program state (with structures representing the run-time heap and stack). In Section 4.6, we define our verification logic and the axiomatic semantics of JML-C specifications. To illustrate how our verification logic would be used to prove the correctness of subclass code using only superclass and subclass specifications, we give some examples of correctness proofs in Section 4.7. In Section 4.8, we conclude with a brief summary and discussion of related work.

4.1 Subclassing Contracts in JML Specifications

In this first section, we review and give an overall description of JML. We start, in subsection 4.1.1, with a brief review of the structure of JML specifications. Then, in subsection 4.1.2, we describe the subclassing contract and the additional information needed by our technique and provided in the callable and accessible clauses. In subsection 4.1.3, we describe the code contract in JML specifications and how it would be used in reasoning.

4.1.1 JML Specifications

In JML, there are two main kinds of specification constructs: those that specify the behavior of methods and those that specify data representations. As described in previous chapters, the behavior of methods is specified through requires, ensures, and assignable clauses; these clauses specify the pre- and postconditions and the side-effects allowed in the implementation of a method. The data representations are specified through the declaration of public model fields together with represents clauses, data group clauses (in and maps clauses), and invariant clauses.

Recall that the modifiers qualifying the keywords that introduce specifications indicate whether the specification that follows is part of the public, protected, or private specification. For example, protected normal_behavior means that the requires, assignable, and ensures clauses, etc. that

```
public class CellContainer {
  //@  public model int cellVal;
  //@  public model int oldVal;
  //@                  in cellVal;

  /*@  public normal_behavior
    @    requires cell != null;
    @    assignable cellVal, oldVal;
    @    ensures cellVal == cell.value && oldVal == cell.value;   @*/
  public CellContainer(IntCell cell);

  /*@  public normal_behavior
    @    requires c != null;
    @    assignable cellVal, oldVal;
    @    ensures cellVal == c.cellVal && oldVal == \old(cellVal)
    @        && \result == Math.abs(cellVal - oldVal);              @*/
   public int set(CellContainer c);

  /*@  public normal_behavior
    @    requires newCell != null;
    @    assignable cellVal, oldVal;
    @    ensures cellVal == newCell.value && oldVal == \old(cellVal)
    @        && \result == Math.abs(cellVal - oldVal);            @*/
   public int set(IntCell newCell);

  /*@  public normal_behavior
    @    assignable \nothing;
    @    ensures \result == Math.abs(cellVal - oldVal);    @*/
  public /*@ pure @*/ int difference();

  /*@  public normal_behavior
    @    assignable \nothing;
    @    ensures \result.value == cellVal;                 @*/
   public IntCell getCell();
}
```

Figure 4.1: Public specification of `CellContainer` in file `CellContainer.jml-refined`.

follow are part of the protected specification for that method and visibility type; this is because of the protected modifier. Public specifications are intended for clients of the class (e.g., Figure 4.1), whereas the public and protected specifications (e.g., Figures 4.1 and 4.2) are intended for customizers, i.e., programmers creating subclasses; the public, protected, and private specifications taken together can be used by programmers implementing or verifying the implementation of the class

```
//@ refines "CellContainer.jml-refined";

public class CellContainer {

  protected IntCell _cell;
  //@                   in cellVal;
  //@                   maps _cell.value \into cellVal;

  protected IntCell _oldCell;
  //@                   in oldVal;
  //@                   maps _oldCell.value \into oldVal;

  //@ protected represents cellVal <- _cell.value;
  //@ protected represents oldVal <- _oldCell.value;

  /*@ protected normal_behavior
    @   requires c != null;
    @   assignable cellVal;
    @   ensures cellVal == c.cellVal && oldVal == c.oldVal;    @*/
  protected CellContainer(CellContainer c);
}
```

Figure 4.2: `CellContainer`'s protected specification in file `CellContainer.jml`.

itself (class `CellContainer` does not have a private specification because there are no private fields or methods).

Also, recall that the modifiers qualifying the keywords that introduce specifications also restrict the scope of variables. For example, protected normal_behavior means that both public and protected variables and methods are in scope in the specification that follows it. Note that, the specification of a protected method or constructor can use public model variables even though protected variables are in scope; this makes its specification independent of the implementation (see Figure 4.2). However, if the public specifications of Figure 4.1 were to access any fields or methods with protected or private visibility, then the JML checker would emit an error.

In the examples in this dissertation, the public, protected, and private specifications have been given in separate files, as in Figures 4.1 and 4.2. However, when a field is declared in more than one file, it must have the same type in each file. For example, the protected field `_x` is declared in the protected specification of Figure 4.2 as well as in the Java file of Figure 4.3 and is the same field.

Similarly, method interfaces must be the same when the same method is declared in more than one file, i.e., when the method name and parameter types are the same, the parameter names[1] and return

type must also be the same. Furthermore, JML specifications specify the method interface as well as its behavior. Therefore, to be correct, each method must implement its specified interface as well as the specified behavior given in the combined public, protected, and private specifications.

As described in subsection 1.4.3, JML allows specifications to be divided into specification cases that make them easier to read and understand. Also, protected specifications may provide some specification cases in addition to those given in the public specification. However, these separate specification cases can be combined into a single specification case with the same meaning [RL03] (see subsection 4.4.3 for more details). Therefore, for simplicity, we will combine specification cases into a single case for use in our proofs and explanations in the sections that follow.

The requires, ensures, assignable, and callable clauses can be omitted from method specifications but when they are, there is an implicit default. For example, when the requires clause is omitted, a method may be called without any restrictions on the pre-state, i.e., the default precondition is `true` in JML. When the assignable (or accessible) clause is omitted, the method can assign to (or access) any location allowed by the Java scope rules, i.e., the default is `\everything`. When the ensures clause is omitted, then nothing is promised about the post-state after method execution, i.e., the default postcondition is `true`. When the callable clause is omitted, then the method can make any call allowed by the Java scope rules, i.e., `\everything` in JML. Similarly, the default invariant clause is `true` which would be conjoined with any inherited invariants (see function *inv* of Figure 4.11).

When the entire method specification is omitted, then overriding methods use the inherited specification; however, when there is no inherited specification, then nothing is promised, i.e., the above defaults are in effect.

Specifiers, customizers, and verifiers are expected to be aware of these defaults. Therefore, a clause must not be omitted if the default has a different meaning than intended. Also, specifiers need to be aware of the consequences of omitting clauses. For example, omitting the callable clause allows all calls; however, this means that such a method, as well as the methods that call it, are likely to be invalidated by new subclasses, e.g., a method `m` cannot super-call a method with no callable clause if methods declared in `m`'s class have additional side-effects or `m`'s class invariant constrains superclass fields (see subsection 5.1.3). This is another reason why we propose tool support to automatically generate the needed callable clauses.

---

1. This requirement makes it easier to combine the separate files into one common specification since no parameter renaming is necessary. The JML tools combine these files when the combined specification is needed, for example, in the run time assertion checker and the JML docs tool. We keep the specifications separate here to emphasize their differing uses by the different kinds of users and to illustrate that not all information would be provided to clients and customizers creating subclasses of library classes.

```
//@ refines "CellContainer.jml";

public class CellContainer {
    protected IntCell _cell;
    protected IntCell _oldCell;

    public CellContainer(IntCell cell) {
        _cell = cell;
        _oldCell = new IntCell(cell.getValue());
    }
    public int set(CellContainer c) {
        return set(c.getCell());
    }
    public int set(IntCell newCell) {
        _oldCell.setValue(_cell.getValue());
        _cell = newCell;
        return difference();
    }
    public /*@ pure @*/ int difference() {
        return Math.abs(_cell.getValue() - _oldCell.getValue());
    }
    public IntCell getCell() {
        return _cell;
    }
    protected CellContainer(CellContainer c) {
        _cell = c._cell;
        _oldCell = new IntCell(c._oldCell.getValue());
    }
}
```

Figure 4.3: `CellContainer`'s implementation in file `CellContainer.java`.

4.1.2 The Subclassing Contract

As described briefly in Chapter 1, the subclassing contract is composed of the callable and accessible clauses. The purpose of the subclassing contract, in our technique, is to give programmers additional information about method behavior so they can avoid downcall and aliasing problems. That is, the information specified in the callable and assignable clauses[2] is needed so the rules given in Chapters 2 and 3 can be enforced modularly; these rules can then be used to prevent the downcall and

2. The accessible clause from the subclassing contract can also be used to allow calls of pure methods in the represents clause, but we leave that as future work.

aliasing problems described in those chapters. Figure 4.4 shows the subclassing contract for class `CellContainer`.

The callable (and accessible) clause in the subclassing contract can be automatically derived from implementation code by the tool described in Chapter 6 (or they can be specified by hand) for all methods and constructors involved in reasoning about downcalls and aliasing. However, the subclassing contract has to be provided for some methods and constructors for which it might not seem obviously needed at first, such as non-public methods and constructors.

In the subclassing contract of Figure 4.4, the \same predicate in the requires clause specifies that this specification case has, effectively, the same precondition as the combined precondition for the given method, i.e., the precondition formed by combining all the specification cases that do not have \same as their precondition. Therefore, \same denotes the disjunction of the non-\same preconditions of all specification cases, for a specific method, in the current class and those inherited from supertypes. For example, in Figure 4.4, \same denotes `true` for method difference() and `cell!=null` for the public constructor. Thus when the precondition is \same, all the clauses that follow apply to all of the other specification cases given for that method. For example, the callable clauses in Figure 4.4 must be satisfied whenever the associated method is executed because their precondition is \same.

A primary reason for the \same precondition is to make it easier to automatically generate the subclassing contract, and in particular, to make it easier to generate the precondition. That is, if the precondition of every specification case in the subclassing contract is \same, then the tool does not have to locate all the specification cases for a method and then create the disjunction of these expressions to form the precondition[3].

4.1.2.1 The Callable Clause

The callable clause lists the methods (and constructors) that the method being specified is allowed to call. Because constructor calls can also cause downcall problems, the term "method" will also include constructors. A method *M directly calls* a method *N* if the code for *M* contains an expression that calls *N*, such as `N()`. If *N* also directly calls method *P*, then *M indirectly calls P*.

Our technique must also handle static overloading, as is permitted in Java. That is, the callable clause must be able to distinguish precisely among a set of overloaded methods (or constructors). Therefore, as is done in Java, this ambiguity is eliminated based on method signatures, i.e., when a method name is overloaded, we require that both the name and formal parameter types be included in the callable clause (see Figure 4.4).

---

3. However, \same can also be used as a convenience in handcrafted specification cases when the specifier does not want to determine the complete precondition for a method. For example, this can be convenient when writing specification cases for overriding methods when the precondition is the same as was specified in the superclass.

```
//@ refines "CellContainer.java";

public class CellContainer {
  /*@ also
    @ protected code normal_behavior
    @   requires \same;
    @   callable new IntCell(int), cell.getValue();    @*/
  public CellContainer(IntCell cell);

  /*@ also
    @ protected code normal_behavior
    @   requires \same;
    @   callable c.getCell(), this.set(IntCell);    @*/
  public int set(CellContainer c);

  /*@ also
    @ protected code normal_behavior
    @   requires \same;
    @   callable IntCell.setValue(int), IntCell.getValue(),
    @         this.difference(); @*/
  public int set(IntCell newCell);

  /*@ also
    @ protected code normal_behavior
    @   requires \same;
    @   callable Math.abs(int), IntCell.getValue(),
    @         IntCell.getValue();  @*/
  public /*@ pure @*/ int difference();

  /*@ also
    @ protected code normal_behavior
    @   requires \same;
    @   callable \nothing;    @*/
  public IntCell getCell();

  /*@ also
    @ protected code normal_behavior
    @   requires \same;
    @   callable new IntCell(int), IntCell.getValue();    @*/
  protected CellContainer(CellContainer c);
}
```

Figure 4.4: Code contract for class `CellContainer` from `CellContainer.refines-jml`.

Also, the specifier should not include unnecessary methods in the callable clause of a method. That is, in general, it is not a good idea for the callable clause to list methods that do not actually need to be called because, for some subclasses, this could unnecessarily invalidate the calling method based on the rules given in Chapter 2. This should not, however, be a problem if the subclassing contract is automatically generated by our tool.

Various kinds of calls must also be distinguished in the callable clause, because the effects on downcall problems can sometimes be subtly different, due to the different semantics of each kind of call. For example, we distinguish two kinds of calls in which the receiver `this` is passed implicitly, i.e., self-calls and super-calls (also described in subsection 1.6.5). In addition, we distinguish three kinds of calls in which `this` in not passed implicitly as the receiver of the call, i.e., static-calls, object-calls, and new object constructor calls.

A *self-call* is a call in which the current receiver object (`this`) is also the receiver of the called method. That is, a call to an instance method of the current receiver, such as `set(u)` which is sugar for `this.set(u)`, is a self-call. So, assuming that the type of `u` is `IntCell`, such self-calls would be allowed if either `this.set(IntCell)` or `set(IntCell)` is listed in the callable clause.

A *super-call* is a call in which the receiver is the built-in pseudovariable `super`, as in `super.set(u)` in Java; to allow this super-call, the callable clause would need to list a signature such as `super.set(IntCell)`, again assuming `IntCell` is the type of `u`. However, a *superclass constructor call*, such as `super(u)` in Java, is also a super-call. Superclass method calls and superclass constructor calls are both included in this definition of super-call because both can be involved in the same kinds of downcall problems for the same reasons and must be reasoned about in the same way. The above superclass constructor call would be allowed if the callable clause lists a super-call such as `super(IntCell)`[4]. Notice also that a downcall during a super-call is a callback because it calls back to a method in the subclass.

The three other kinds of calls that we need to distinguish in the callable clause either do not have a receiver or they have a different receiver than the calling method. As mentioned earlier, these three kinds of calls are object-calls, static-calls, and new object constructor calls.

Our technique needs to know when there are object-calls, static-calls, and new object constructor calls because the calling method's receiver can be passed as an explicit argument in these calls. These kinds of calls can, therefore, indirectly access the calling method's receiver and use it to make downcalls. However, by “explicit” we do not mean that `this` must literally be passed as an actual argument; calls in which `this` is passed inside a data structure, such as an array or object, are also considered to be this-argument calls. Thus a *this-argument call* is one that passes the caller’s receiver as an actual parameter or inside an argument object. Therefore, the caller’s receiver is aliased and

4. In languages with multiple-inheritance, like C++, one could also refer to a specific superclass in a super-call. For example, if the super-call `CellContainer::set(u)` is to be allowed, then one would list `CellContainer::set(IntCell)` in the callable clause of a C++ program.

reachable directly through a formal parameter or indirectly through a field of an argument object. Note that the argument object in a this-argument call could be the receiver of an object-call; thus the caller's receiver could be aliased through a field of the object-call's receiver. Therefore, our technique must handle the possibility of such aliasing when there are side-effects (see also subsection 5.1.3.1).

We also need to know whether a method makes object-calls because, in the presence of aliasing, an object-call could be a self-call or a this-argument call. Recall that an *object-call* is a call in which the receiver is an object other than `super` or `this`, e.g., `p.set(u)` is an object-call. In a method with `p` as a formal parameter, such an object-call would be allowed if `CellContainer.set(IntCell)` were listed in the callable clause (assuming `p` has static type `CellContainer` and `u` has type `IntCell`).

Another kind of call that we need to distinguish is the static-call. A *static-call* is a call to a static method, i.e., a method declared with the static modifier. Static methods have no implicit receiver and, therefore, cannot make self-calls. For example, the call `Math.abs(x)` is a static-call because `Math` is a built-in class in Java and not a receiver object. Notice, however, that the specification of such static-calls will look the same as an object-call on a field or local variable (as described above). However, we can distinguish object-calls from static-calls using the static attribute of the called method, i.e., through the presence or absence of the static modifier in the method declaration. If the called method is not static, then we know it is an object-call. However, for simplicity, we will not consider static-calls in this dissertation (see assumptions in subsection 1.6.6). Nonetheless, because static methods cannot be overridden, they could be handled as described in subsection 2.6.4 for private methods. That is, static-calls could be handled by not requiring that they be listed in the callable clause; however, the indirect, non-static calls made during a static-call would have to be listed. Furthermore, modular checking would require that a static method's callable clause list all of its direct and indirect object-calls (there would be no self-calls or super-calls).

We also need to know when there are new object constructor calls. A *new object constructor call* is an expression that creates a new object by invoking a constructor; in Java and C++, `new CellContainer(c)` invokes a constructor. If the callable clause has a signature such as `new CellContainer(IntCell)`, then a method is allowed to invoke that new object constructor (assuming `c` has type `IntCell`). The keyword new is required because a method can be declared with the same name as its enclosing type. For example, inside class `CellContainer`, if a callable clause lists `CellContainer(IntCell)`, without the new, then a call to a method named `CellContainer` would be allowed but calls to a constructor with that signature would not be allowed.

Precisely identifying self-calls assumes perfect knowledge of aliasing; that is, one may only be able to decide at run-time when a self-call is being made by a specific piece of program text. However, when the tool is checking or statically generating the callable clause, it only has approximate knowledge of aliasing. To compensate, an object-call `T.set` that appears in a callable clause will be

considered a self-call whenever `T` is the type (or supertype) of the calling method's receiver (see subsection 5.1.3.1). Similarly, when the type of the formal parameter of the object-called method is the type (or supertype) of the calling method's receiver, then that object-call will be considered a this-argument call.

We also need to know about these different kinds of calls so we can detect possible callback cycles (see Section 2.5). However, for simplicity as explained in subsection 5.1.5, we will not formalize the callback cycle rules. We leave that as future work.

4.1.2.2 The Accessible Clause

The accessible clause of a method *M* specifies the fields that *M* is allowed to directly or indirectly access (read). Accesses to instance variables of the receiver (`this`) are called *self-accesses* and accesses to fields of objects other than the receiver are called *object-accesses* (these are analogous to self-calls and object-calls). The different kinds of field accesses need to be distinguished in the accessible clause for the same reasons that we need to distinguish between the different kinds of call in the callable clause.

An important purpose of the accessible clause is to prevent constructors from making downcalls to methods that access subclass fields that have not yet been initialized (see subsection 2.4.5). The other use is when determining the fields accessed when a method is called in the represents clause (see subsection 3.3.3). However, the need to use the accessible clause can be eliminated by using a more restrictive invalidation rule (subsection 5.1.3) and a more restricted syntax for the represents clause (subsection 4.3). Therefore, to simplify the formal system presented in this chapter, we leave as future work the formalization of the weaker (and more flexible) versions of the rules as given in subsections 2.4.5 and 3.3.3; they will need to be formalized through the use of the accessible clause (similar to our rules that use the callable and assignable clauses in the formal system presented in Chapter 5).

4.1.3 Code Contracts

The *code contract* is composed of the specification cases modified by the keyword code (see Figure 4.4). The code contract specifies properties of the method's implementation and is inherited in the same way as the superclass code, that is, it is inherited without change unless the superclass method is overridden. Whenever a superclass method is overridden, its code contract is not inherited by the subclass. Therefore, unlike other specification cases, the code contract can be replaced to reflect the properties of the overriding subclass method. In summary, clauses in the code contract are specific to an implementation and are not inherited by subclasses whereas clauses that occur in other specification cases are inherited and must be satisfied by subtypes (see subsection 1.4.1). Also, abstract methods (including methods of interface types) cannot have a code contract because they have no implementation code.

Recall also (subsection 1.4.1) that an overriding method must satisfy the inherited supertype specification as well as any additional specification cases given in the subtype. However, the inherited specification does not include the specification cases in the code contract. That is, supertype specification cases in the code contract are not inherited by overriding subtype methods, because the code contract specifies the behavior of the implementation code in the type where that specification is given. Therefore, an abstract method cannot have a code contract since abstract methods do not have implementation code. Furthermore, the specification cases in the code contract have a different inheritance semantics than those occurring in the rest of the specification, i.e., the code contract is not inherited by overriding methods whereas specification cases in the rest of the specification are always inherited by overriding subtype methods.

In summary, when a superclass method is not overridden, then that method's code and its code contract are inherited without change from the superclass. However, when a method is overridden, then its code contract must be specified in the subclass, and this new code contract supersedes the one given in the superclass. Therefore, a correct implementation of a subclass method or constructor must satisfy the code contract given in the subclass and not necessarily the code contract provided in the superclass. For example, the code contract in Figure 4.4 only applies to methods in `CellContainer`, not to overriding methods in any new subclasses of `CellContainer`. If no code contract is specified for a subclass method, then the default specification (subsection 4.1.1) becomes the code contract.

It may not be possible to specify all calls allowed in the public callable clause because not all methods are visible in the public specification, e.g., self-calls and super-calls to protected methods will not be in scope in a public specification. Therefore, our technique provides a way to specify the callable methods that are not be visible in the public specification. For example, specifying `this.*` in a public callable clause allows all self-calls; similarly, `super.*` allows all super-calls. Also, as in our examples, the callable clause can be omitted to allow all possible calls, i.e., the default is `\everything`. However, the specifier (or tool) should also provide a callable clause in the code contract to reduce the allowable calls to only those methods actually called in the implementation; the code contract (which is usually part of the protected specification so it is visible to customizers) would be combined with the public callable clause.

Thus the public callable clause and protected code contract are combined and used in our reasoning technique. For example, the public callable clauses are omitted in Figure 4.1; thus they, by default, allow all self-calls and super-calls. However, this is reduced in the code contracts given in Figure 4.4, i.e., only the calls actually made in the method implementations are listed. Note also that the callable methods in the code contract must be a subset of those allowed by the public specification (see also subsection 5.1.7).

Our technique is only concerned with locating self-calls and super-calls that may make downcalls. Therefore, specifying object-calls in the callable clause is only necessary because of aliasing, i.e., if an

object-call could be a self-call (based on the types of the receivers), then we assume that it is and apply our rules accordingly; this is a conservative approach because when an object-call is not a self-call, our rules would not apply (see subsection 5.1.3). The advantage of allowing all self-calls and super-calls in the public specification is that it can be replaced by the code contract. Therefore, based on the callable clause in the code contract, customizers and verifiers can reason about possible self-calls and super-calls that reflect the implementation code. However, if the code contract changes in a superclass, then all subclasses have to be re-checked and reverified.

4.2 The Java-C Syntax

In this subsection, we introduce a core subset of the Java programming language; we will refer to this subset as Java-C. The language includes features common to most OO programming languages, such as Java, but, for simplicity, we have eliminated certain language specific features and features that can be simulated by other constructs.

A Java-C program is a set of class declarations (the syntax does not include interfaces). A class declaration gives the class name its superclass name and member declarations. A member declaration is either a declaration of an instance field or instance method; static fields and static methods are not allowed. Fields can either be public model fields or protected concrete fields; private fields are not allowed. Also, for simplicity, we do not allow field initializers; fields have to be initialized through an assignment in a constructor.

Java-C includes built-in Java types int and boolean and user-defined class types; the singleton null type is a subset of all user-defined types. The superclass-subclass relationship is declared, as in Java, with the superclass name following the extends keyword.

For simplicity, the syntax of method signatures and method calls is restricted to one parameter; nonetheless, if our technique can handle one parameter, then it can be extended to handle more than one argument in the same way.

Also, to simplify our definition of the axiomatic semantics of Java-C (subsection 4.6), expressions in Java-C do not have side-effects. For example, method calls are not allowed in expressions[5]; nonetheless, method calls in expressions can be simulated. That is, the syntax allows the result of a method call (that may also have other side-effects) to be assigned to a variable, and that variable can then be accessed in expressions.

There are three production rules for method calls since the semantics and handling of self-calls, super-calls, and object-calls need to be slightly different. Thus each of these kinds of call has a separate production rule to make incorporating the rules from Chapters 2 and 3 easier.

---

5. Calls of pure methods could be allowed since such expressions are referentially transparent, but we leave them out to help keep our language and its semantics more concise, i.e., with fewer special cases that are unrelated to the problems our technique needs to handle.

---

program ::= *class_decl**
class_decl ::= *modifier** `class` *id* `extends` *id* `{` *member_decl** `}`
modifier ::= `public` | `protected` | `abstract` | `model` | `pure`
member_decl ::= *field_decl* | *method_decl* | *jml_data_rep_spec*
field_decl ::= *f_mods* *type* *id* `;` *jml_data_group_clause**
f_mods ::= `public model` | `protected`
type ::= `boolean` | `int` | *id*
method_decl ::= *jml_method_spec* *modifier** *method_signature* *body*
method_signature ::= *type* *id* `(` *formal* `)` | `void` *id* `(` *formal* `)`
formal ::= *type* *id*
body ::= `;` | *block*
block ::= `{` *stmt* `}`
stmt ::= `;` | *assign_stmt* | *method_call* `;` | *block*
| `while` `(` *expr* `)` *stmt* | `if` `(` *expr* `)` *stmt* `else` *stmt*
| *local_decl* | `return` *expr* `;` | `super` `(` *expr* `)` `;` | *stmt* *stmt*
assign_stmt ::= *var_ref* = *expr* `;` | *var_ref* = *method_call* `;`
| *var_ref* = `new` *id* `(` *expr* `)`
local_decl ::= *type* *id* `;`
expr ::= `this` | *var_ref* | *literal* | `(` *type* `)` *expr* | `(` *expr* `)` | *expr* *bop* *expr* | *uop* *expr*
bop ::= **`+`** | **`-`** | **`*`** | **`/`** | **`%`** | **`||`** | **`&&`** | **`<`** | **`>`** | **`<=`** | **`>=`** | **`==`** | **`!=`**
uop ::= **`!`** | **`-`**
var_ref ::= *id* | `this` `.` *id*
method_call ::= `this` `.` *id* `(` *expr* `)` | `super` `.` *id* `(` *expr* `)` | *var_ref* `.` *id* `(` *expr* `)`
literal ::= `true` | `false` | *int_literal* | `null`

Figure 4.5: The syntax of the Java-C subset of Java used in this chapter. The tokens *id* and *int_literal* are the usual Java identifier and integer literals and are not defined here. Also, *jml_data_rep_spec*, *jml_data_group_clause*, and *jml_method_spec* are defined in Figure 4.6.

Expressions are boolean or integer literals, the `null` literal, the receiver `this`, or a variable reference; they also include cast expressions like `(T)ref` and most of the standard binary (*bin_op*) and unary (*un_op*) operators. Note, however, that model fields are accessible in specifications, but they are not visible in method statements.

In accordance with our assumptions, the syntax of Java-C does not include arrays, exceptions, interfaces, nested types, anonymous types, and static and package visible variables and methods. However, we have included the features involved in the problems that need to be handled by our

technique, i.e., the problems related to aliasing, code inheritance, and subclassing in single-dispatch OO programming languages. For example, in Java-C, all fields can be modified (if the specification allows it) and all methods can be overridden since, in accordance with our assumptions, we do not allow static, final, or private fields and methods. Such fields and methods are not allowed in Java-C for simplicity and because they can be considered special cases of the aliasing and downcall problems that are handled by the rules described in Chapters 2 and 3; we also make this assumption because private fields and unoverrideable methods can result in additional unimplementable subclasses (see subsection 2.9.3).

The Java-C syntax in Figure 4.5 also specifies where the JML specifications are located in a class declaration, e.g., where the invariant and represents clauses (*jml_data_rep_decl*), the in and maps clauses (*jml_data_group_clause*), and the normal_behavior clauses (*jml_method_spec*) are located. The syntax of these constructs is given in Figure 4.6 and is described in Section 4.3 below. Note, however, that JML specifications, including keywords and modifiers, such as model and pure, have to be in annotation comments as shown in our examples.

4.3 The JML-C Syntax

We now introduce a core subset of the Java Modeling Language (JML); we will refer to this subset as JML-C. JML-C is a notation for specifying the interface and behavior of Java-C classes and methods (see also Section 1.3). For simplicity, we have eliminated those features of JML related to constructs omitted from Java-C, such as, exceptions and arrays. However, JML-C includes the core features (of JML) needed to demonstrate how our technique works.

The syntax of JML-C is specified in Figure 4.6. Included is the syntax of the invariant and represents clauses; these clauses specify the properties of the data fields of a class. Recall from Chapter 1 that these clauses are prefixed by modifiers that specify their visibility, e.g., public or protected. The syntax of the right hand side of the represents clause, *rep_expr*, is a subset of the expressions allowed in Figure 3.10[6]; thus the right side will obey the Represents Clause Access Rule given in subsection 3.3.3. The maps and in clauses specify the data group memberships of newly declared fields.

The requires clause specifies the precondition that must hold on entry to a method; the ensures clause specifies the postcondition that must hold on exit. A *predicate* is a boolean expression, i.e., a subset of the Java expressions. The assignable clause specifies the frame axiom, that is, the fields assignable during the method's execution. Similarly, the callable clause specifies the methods that may be called during the method's execution. More of the specific details and semantics are given in Chapter 1 and in JML's axiomatic semantics defined in Section 4.6.

---

6. Method calls are not allowed since that would require the use of the accessible clause.

jml_data_rep_decl ::= *modifier invariant_clause* | *modifier represents_clause*
jml_data_group_clause ::= *in_clause* | *maps_into_clause*
jml_method_spec ::= *modifier* `normal_behavior` *requires_clause spec_body*
invariant_clause ::= `invariant` *predicate* ;
represents_clause ::= `represents` *id* `<-` *rep_expr* ;
rep_expr ::= `this` **.** *id* | `this` **.** *id* **.** *id* | *literal* | *rep_expr bin_op rep_expr* | *un_op rep_expr*
in_clause ::= `in` *id_list* ;
maps_into_clause ::= `maps` *id* **.** *id* `\into` *id_list* ;
id_list ::= *id* | *id* **,** *id_list*
spec_body ::= *assignable_clause ensures_clause callable_clause*
requires_clause ::= `requires` *predicate* **;**
assignable_clause ::= `assignable` *store_ref_list* **;**
ensures_clause ::= `ensures` *predicate* **;**
callable_clause ::= `callable` *method_name_list* **;**
store_ref_list ::= *store_ref* | *store_ref* **,** *store_ref_list*
store_ref ::= `this` **.** *id* | *id* **.** *id*
method_name_list ::= *method_name* | *method_name* , *method_name_list*
method_name ::= `this` **.** *call_signature* | `super` **.** *call_signature*
| *id* **.** *call_signature* | *type* **.** *call_signature*
call_signature ::= *id* (*type*) | *id*

Figure 4.6: Syntax of the JML-C subset used in this chapter.

4.4 Type Environments

Our additional static checking rules (Section 5.1), as well as the operational and axiomatic semantics (Sections 4.5 and 4.6), use information gathered during the standard Java type checking. Therefore, we assume that each program has been checked and does not violate any of the rules of the Java type system. We further assume that the type checker has created a type environment, i.e. `TEnv`, containing all bindings for type, field, and method names referenced or declared in a program. For simplicity (and because it does not affect the soundness of our technique), we also assume that type names do not conflict, e.g., we do not have packages or qualified type names. Similarly, for simplicity, we assume that method names are not overloaded and that fields are not declared in a subclass with the same name as a field inherited from a superclass.

The type environment (`TEnv`) is specific to a particular program and contains the information shown in Figure 4.7. The domain of type environments (*TypeEnv*) describes mappings from a type

Identifiers:

VarId = the set of valid variable names
MethId = the set of valid method names
TypeId = the set of declared type names
ObjId = the set of allocated object id's

Types:

Type = {`BoolT`, `IntT`, `NullT`} ∪ *TypeId*
typeOf: *Value* → *Type*
typeOf: *expr* → *Type*
_ ≤ _: *TypeId* × *TypeId* → *BVal*
_ < _: *TypeId* × *TypeId* → *BVal*
superOf: *TypeId* → *TypeId*

Type Environments:

TypeEnv = *TypeId* → *TypeDecl* ∪ {`undef`}
TEnv: *TypeEnv*

Type Declarations:

TypeDecl: *TypeId* × *fieldTable* × *methodTable* × *expr* × *RepDecls*
superclassOf: *TypeDecl* → *TypeId*
fieldsOf: *TypeDecl* → *fieldTable*
methodsOf: *TypeDecl* → *methodTable*
invOf: *TypeDecl* → *expr*
repOf: *TypeDecl* → *RepDecls*
fieldTable: *VarId* → *VarDecl* ∪ {`undef`}
methodTable: *MethId* → *MethDecl* ∪ {`undef`}
RepDecls: *VarId* → *expr*
lookupField: *TypeId* × *VarId* → *VarDecl* ∪ {`undef`}
lookupMethod: *TypeId* × *MethId* → *MethDecl* ∪ {`undef`}
whereMethodDecl: *TypeId* × *MethId* → *TypeId*

Figure 4.7: Types and the type environment created during type checking of a specific program.

name (*TypeId*) to a type declaration (*TypeDecl*), i.e., the map will contain a binding for every class declared or referenced in a program.

4.4.1 Type Declarations

A *TypeDecl* contains the information from a class declaration, i.e., the superclass declared in the `extends` clause, lookup tables for the fields and methods declared in this class, and the data

Field and Variable Declarations:

$$\begin{aligned}
&\mathit{VarDecl}: \mathit{Modifiers} \times \mathit{Type} \times \mathit{VarId} \times \mathit{MapsDecl} \times \mathit{InDecl}\\
&\mathit{typeOf}: \mathit{VarDecl} \rightarrow \mathit{Type}\\
&\mathit{mapsOf}: \mathit{VarDecl} \rightarrow \mathit{MapsDecl}\\
&\mathit{isModelField}: \mathit{VarDecl} \rightarrow \mathrm{BVal}\\
&\mathit{inOf}: \mathit{VarDecl} \rightarrow \mathit{InDecl}\\
&\mathit{MapsDecl} = \{\, (f.x, g) \mid g \in \mathit{VarId}, x \in \mathit{VarId}, g \in \mathit{VarId} \,\}\\
&\mathit{InDecl} = \{\, g \mid g \in \mathit{VarId} \,\}
\end{aligned}$$

Method Declarations:

$$\begin{aligned}
&\mathit{MethDecl}: \mathit{Type} \times \mathit{VarDecl} \times \mathit{stmt} \times \mathit{AssignSet} \times \mathit{CallSet}\\
&\mathit{parmOf}: \mathit{MethDecl} \rightarrow \mathit{VarDecl}\\
&\mathit{bodyOf}: \mathit{MethDecl} \rightarrow \mathit{stmt}\\
&\mathit{reqOf}: \mathit{MethDecl} \rightarrow \mathit{expr}\\
&\mathit{ensOf}: \mathit{MethDecl} \rightarrow \mathit{expr}\\
&\mathit{assignsOf}: \mathit{MethDecl} \rightarrow \mathit{AssignSet}\\
&\mathit{codeAssignsOf}: \mathit{MethDecl} \rightarrow \mathit{AssignSet}\\
&\mathit{callsOf}: \mathit{MethDecl} \rightarrow \mathit{CallSet}\\
&\mathit{codeCallsOf}: \mathit{MethDecl} \rightarrow \mathit{CallSet}\\
&\mathit{AssignSet} = \{\, \texttt{this}.g \mid g \in \mathit{VarId} \,\} \cup \{\, \texttt{p}.g \mid g \in \mathit{VarId} \,\}\\
&\qquad\qquad \cup \{\, e.g \mid e \in \mathit{expr} \wedge g \in \mathit{VarId} \,\}\\
&\mathit{CallSet} = \{\, \texttt{this}.m \mid m \in \mathit{MethId} \,\} \cup \{\, \mathrm{T}\mathbf{::}m \mid \mathrm{T} \in \mathit{TypeId},\ m \in \mathit{MethId} \,\}\\
&\qquad \cup \{\, \mathrm{T}.m \mid \mathrm{T} \in \mathit{TypeId},\ m \in \mathit{MethId} \,\}\\
&_\,[\,_ \leftarrow _\,]: \mathit{AssignSet} \times \mathit{Parm} \times \mathit{expr} \rightarrow \mathit{AssignSet}\\
&\mathit{Parm} = \{\, \texttt{this}, \texttt{p} \,\}\\
&\mathit{getParmType}: \mathit{TypeId} \times \mathit{MethId} \rightarrow \mathit{Type}\\
&\mathit{getBody}: \mathit{TypeId} \times \mathit{MethId} \rightarrow \mathit{stmt}
\end{aligned}$$

Figure 4.8: Field and method declarations allowed in classes.

specifications from the invariant (*expr*) and represents clauses (*RepDecl*) (see *JmlDataSpecs* in Figure 4.6). The function *superclassOf* extracts the superclass name from a type declaration. Similarly, functions *invOf* and *repOf* project the data specifications from the invariant and the represents clauses declared in that class. Each type declaration (*TypeDecl*) has a *fieldTable* and *methodTable* containing the bindings for the fields and methods declared in that specific class; they map the field or method name to the associated field or method declaration. These lookup tables for fields and methods are obtained from a *TypeDecl* by the *fieldsOf* and *methodsOf* functions. The functions *lookupField* and

---

superOf(T) = **if** T = `Object`
then `undef`
else *superclassOf*(`TEnv`(T))

lookupField(T, *f*) = **if** *fieldsOf*(`TEnv`(T)) (*f*) = `undef`
then if *superOf*(T) = `undef`
then `undef`
else *lookupField*(*superOf*(T), *f*)
else *fieldsOf*(`TEnv`(T)) (*f*)

lookupMethod(T, *m*) = **if** *methodsOf*(`TEnv`(T)) (*m*) = `undef`
then if *superOf*(T) = `undef`
then `undef`
else *lookupMethod*(*superOf*(T), *m*)
else *methodsOf*(`TEnv`(T)) (*m*)

whereMethodDecl(T, *m*) = **if** *methodsOf*(`TEnv`(T)) (*m*) ≠ `undef`
then T
else *whereMethodDecl*(*superOf*(T), *m*)

getBody(T, *m*) = *bodyOf*(*lookupMethod*(T, *m*))

getParmType(T, *m*) = *typeOf*(*parmOf*(*lookupMethod*(T, *m*)))

Figure 4.9: Auxiliary functions used in the type checking and semantics of Java-C and JML-C.

---

lookupMethod search for specific field and method declarations in the class hierarchy. That is, *lookupField* and *lookupMethod* are necessary for looking up the inherited fields and methods visible in a particular context; these and other auxiliary functions that use the type environment are defined in Figure 4.9.

4.4.2 Field Declarations

Field and variable declarations are represented by the *VarDecl* data structure containing the modifiers, type and data group declarations (*MapsDecl* and *InDecl*). The projection functions are *typeOf*, *mapsOf*, and *inOf*. The predicate *isModelField* determines, from the modifiers, whether or not a field is a model field.

```
S ≤ T = if S = T
        then true
        else if superOf(S) = undef
             then false
             else superOf(S) ≤ T

S < T = if S = T then false else S ≤ T
```

Figure 4.10: Auxiliary functions used in the type checking and semantics of Java-C and JML-C.

As described in subsection 2.2.1, data group declarations (in and maps clauses) specify dependency relationships between model fields and concrete fields; they also specify the concrete fields that can be assigned when a model field is allowed to change. Note also that fields can have data group declarations but parameters and local variables cannot (see syntax in Figure 4.5). *InDecl* is the set of model fields (data groups) that depend on the declared field; it is derived from the in clause(s) of a field declaration. *MapsDecl* is a set of pairs that specifies the dynamic dependencies between the fields ($f.x$) of an internal object (f) and the model fields (g) that depend on them; it is derived from the maps clauses of the declared field (f).

4.4.3 Method Declarations

Method declarations are represented by the *MethDecl* data structure containing the return type, method name, parameter declaration, method body, and JML method specification (requires, ensures, assignable, and callable clauses). The projection functions are *parmOf*, *bodyOf*, *reqOf*, *ensOf*, *assignsOf*, *callsOf*, and *codeCallsOf* for retrieving the parameter, body, precondition, postcondition, assignable fields, and callable methods from a specific method declaration. The *callsOf* function extracts the set of callable methods from the public and protected specifications for a method as declared in a specific type. The *codeCallsOf* function extracts this set from the code contract described in subsection 4.1.3.

We assume that multiple specification cases have been combined into a single case with one requires clause (precondition), one ensures clause (postcondition), and one assignable clause (frame axiom). Therefore, the projection functions *reqOf*, *ensOf*, and *assignsOf* retrieve these combined components from a method specification. However, the functions *req*, *ens*, and *assigns* (Figure 4.11) are also needed to combine the superclass and subclass specification cases into a single case (see subsection 4.4.4).

```
inv(T) = if superOf(T) = undef
         then invOf(TEnv(T))
         else invOf(TEnv(T)) && inv(superOf(T))

req(T, m) =
         if superOf(T) = undef ∨ lookupMethod(superOf(T), m) = undef
         then reqOf(lookupMethod(T, m))
         else if methodsOf(TEnv(T)) (m) = undef
              then req(superOf(T), m))
              else reqOf(lookupMethod(T, m)) || req(superOf(T), m))

ens(T, m) =
         if superOf(T) = undef ∨ lookupMethod(superOf(T), m) = undef
         then postCond(lookupMethod(T, m))
         else if methodsOf(TEnv(T)) (m) = undef
              then ens(superOf(T), m)
              else postCond(lookupMethod(T, m)) && ens(superOf(T), m))

postCond(M) = (! \old(reqOf(M)) || ensOf(M) )

assigns(T, m) =
        if superOf(T) = undef ∨ lookupMethod(superOf(T), m) = undef
        then assignsOf(lookupMethod(T, m))
        else if methodsOf(TEnv(T)) (m) = undef
             then assigns(superOf(T), m)
             else assignsOf(lookupMethod(T, m)) ∩ assigns(superOf(T), m)

calls(T, m) = codeCallsOf( lookupMethod(T, m) )
```

Figure 4.11: Auxiliary functions for combining superclass and subclass specification cases.

For simplicity, we assume that only one formal parameter, named `p`, is declared in each method. However, our technique can be extended to multiple parameters by handling them in the same way as a single parameter. That is, if our technique is sound for one pair of parameters (the formal parameter and receiver), then the soundness proof can be extended in the same way to each pair of parameters in a method call.

4.4.4 Extracting Specifications from Class and Method Declarations

Functions for extracting the type invariant from class declarations, and the pre- and postconditions from method declarations are given in Figure 4.11. The function *inv*(T) extracts and combines the type invariant from each of the class declarations in the class hierarchy; this function transforms the syntax tree into a combined expression tree; it does not evaluate the invariant expression or determine the validity of that invariant property. Similarly, the functions *req*(T, *m*) and *ens*(T, *m*) of Figure 4.11 extract and combine the pre- and postconditions, respectively, from method declarations in the class hierarchy; these functions do not evaluate these assertions (see also subsection 5.1.7).

The functions *assigns* and *calls* of Figure 4.11 extract the assignable fields and callable methods for a specific method. The function *assigns*(T, *m*) calculates the set of assignable fields from the class hierarchy of T for the given method *m*. Notice that it combines the superclass and subclass assignable clauses through set intersection; thus subclasses are only allowed to assign to fields in groups permitted by the superclass specification[7]. Similarly, the function *calls*(T, *m*) extracts the callable methods from the current code contract using *codeCallsOf*.

4.4.5 Logical Expressions

We assume that logical expressions in our function definitions and semantic rules are evaluated (as in Java) using the short circuit (or minimal) evaluation strategy. That is, subexpressions of a logical expression are evaluated from left to right until the value of the entire expression is known. For example, if the first disjunct (conjunct) evaluates to true (false), then the rest of the disjuncts (conjuncts) do not have to be evaluated. This strategy eliminates the need to define a special function to accomplish a short circuit evaluation. For example, "*e1* || *e2*" would have to be rewritten as "**if** *e1* **then** `true` **else** *e2*".

4.5 Operational Semantics of Java-C

Our operational semantics has two components, the model of the program state (Figures 4.12 to 4.15) and the execution rules defined inductively over the syntax constructs of Java-C (Figures 4.16 to 4.19). The program state is modeled, in Figure 4.13, as a pair of object stores (*ObjStore*). The first object store represents stack storage and the second heap storage. As usual, the stack contains storage for local variables and parameters and the heap for objects created with a new object constructor call. An *ObjStore* maps variable locations to the value stored at that location (*Location* → *Value*).

---

7. This is not true in general because a subclass could have a specification case that does not overlap with the precondition of the superclass specification; that is, a behavioral subtype would be allowed to assign to additional fields in non-overlapping specification cases. However, for simplicity we will not consider such possibilities in our formalization here since determining whether or not a specification case overlaps requires a proof rather than being checkable statically using assignable clauses.

Value Domains:

$BVal = \{\ boolV(b) \mid\ b \in \{true, false\}\ \}$
$IVal = \{\ intV(v) \mid\ v \in \{MinInt, \ldots -1, 0, 1, \ldots MaxInt\}\ \}$
$Null = \{\ voidV(null)\ \}$
$Value = BVal \cup IVal\ \cup Null \cup ObjectRef$

Values:

$default: Type \rightarrow Value$
$val: literal \rightarrow Value$
$apply: bop \times Value \times Value \rightarrow Value$
$apply: uop \times Value \rightarrow Value$

Object References:

$ObjectRef = \{\ ref(T, OI) \mid\ T \in TypeId,\ OI \in ObjId\ \}$
$refType: ObjectRef \rightarrow TypeId$
$objId: ObjectRef \rightarrow ObjectId$
`global`: *ObjectRef*
`local`: *ObjectRef*

Locations:

$Location = \{\ loc(x, r) \mid\ x \in VarId, r \in ObjectRef\ \}$
$varId: Location \rightarrow VarId$
$objRef: Location \rightarrow ObjectRef$
$refType: Location \rightarrow TypeId$
$objId: Location \rightarrow ObjectId$
$locType: Location \rightarrow Type \cup \{\texttt{undef}\}$
`objIdLoc`: *Location*
`thisLoc`: *Location*
`resultLoc`: *Location*

Figure 4.12: Value domains and locations used in the operational semantics of Java-C.

Locations contain the value of a field (Figure 4.12). However, locations need to distinguish the values of fields in one object from the values of those same fields in a different object. Thus a location is modeled as a variable name and an object reference. An object reference (*ObjectRef*) contains the type of the object (needed for dynamic binding) and the object-id. The object-id uniquely identifies a specific object and is allocated by a new object constructor call. The run-time type of an object is extracted using the function *refType* and the object-id is retrieved by function *objId* of Figure 4.14. The type of the value stored in a field location is obtained through the *locType* function (Figure 4.14).

Object Storage:

ObjStore = *Location* → *Value* ∪ {`undef`}
_ [_ := _]: *ObjStore* × *Location* × *Value* → *ObjStore* ∪ {`undef`}
`emptyS`: *ObjStore*

The Program State:

State = { *state*(s, h) | s ∈ *ObjStore*, h ∈ *ObjStore* }
stackOf: *State* → *ObjStore*
heapOf: *State* → *ObjStore*
getValue: *State* × *Location* → *Value* ∪ {`undef`}
_ [_ := _]: *State* × *Location* × *Value* → *State* ∪ {`undef`}
new: *TypeId* × *State* → (*ObjectRef* × *State*) ∪ {`undef`}
getObjId: *State* → *ObjectId*
nextObjId: *State* → *ObjStore*
successor: *IVal* → *IVal*
`initialState`: *State*

Figure 4.13: The model of a program state and the related function signatures used in the operational semantics of Java-C.

These object-id's have to be unique for each allocated object. To ensure this, the value of the next object-id is stored in a field (`\next`) of a special global object in the heap (the object reference of this unique global object named `global` is defined in Figure 4.14). Every time a new object is allocated by the *new* operator, the value of the next object-id is increased. The functions *getObjId* and *nextObjId*, defined in Figure 4.15, are used in the operational semantics to maintain the value of this field (its unique location named `objIdLoc` is defined in Figure 4.14). We also store the return value of a method in field `\result` of this global object (see `resultLoc` defined in Figure 4.14). The value of this field has the proper type because of standard type checking done prior to execution.

Stack based variables are modeled as fields of a single object in an object store separate from heap storage. This single stack object is referenced using a unique object reference named `local` defined in Figure 4.14; it represents the stack frame. For example, the location of a local variable or parameter `x` is represented by *loc*`(x,local)`. Even though this local object reference is the same for every frame (i.e., `local`), these local variables do not conflict with other stack variables because a new, empty object store is always allocated for the stack at the beginning of each call (see Figure 4.18 and function `emptyS` of Figure 4.15). An empty object store is modeled by the function `emptyS` defined in Figure 4.15.

Values:

default(BoolT) = `false`
default(IntT) = 0
default(*TypeId*) = `null`
mkVal(*lit*) = **if** *lit* = `null` **then** *voidV*(null)
else if *lit* ∈ {`true`, `false`} **then** bool*V*(*lit*)
else *intV*(*lit*)

Object References:

refType(*ref*(T, OI)) = T
objId(*ref*(T, OI)) = OI
`global` = *ref*(`\Global`, *intV*(0))
`local` = *ref*(`\Stack`, *intV*(1))

Locations:

varId(*loc*(x, r)) = x
objRef(*loc*(x, r)) = r
refType(*loc*(x, r)) = refType(r)
objId(*loc*(x, r)) = *objId*(r)
locType(L) =
if `TEnv`(*refType*(L)) = `undef`
then `undef`
else if *lookupField*(*refType*(L), *varId*(L)) = `undef`
then `undef`
else *typeOf*(*lookupField*(*refType*(L), *varId*(L)))
`objIdLoc` = *loc*(`\next`, `global`)
`thisLoc` = *loc*(`this`, `local`)
`resultLoc` = *loc*(`\result`, `global`)

Figure 4.14: The definition of the domain functions used in the operational and axiomatic semantics of Java-C.

Our model does not simulate some implementation features of Java or hardware limitations, such as garbage collection, stack overflow, or memory capacity overflow. We also include a limited set of values. That is, the set of data values is modeled as the disjoint union of the boolean and integer values, *null*, and the set of object references (*ObjectRef*).

In addition, we assume that the program has been type checked and that the required information specific to that program is contained in the type environment `TEnv`. A type environment, shown in

Object Storage:

OS[L := V] = λ `loc` . **if** `loc`=L **then** V **else** OS(`loc`)

`emptyS` = λ `loc` . **if** *locType*(`loc`) = `undef`
then `undef`
else *default*(*locType*(`loc`))

The Program State:

stackOf(*state*(s, h)) = s
heapOf(*state*(s, h)) = h
getValue(S, L) = **if** *objRef*(L)=local
then *stackOf*(S)(L)
else *heapOf*(S)(L)
S[L := V] = **if** *objRef*(L)=local
then *state*(*stackOf*(S)[L := V], *heapOf*(S))
else *state*(*stackOf*(S), *heapOf*(S)[L := V])
new(T, S) = (*ref*(T, *getObjId*(S)), *state*(*stackOf*(S), *nextObjId*(S)))
getObjId(S) = *heapOf*(S)(`objIdLoc`)
nextObjId(S) = *heapOf*(S)[`objIdLoc` := *successor*(*getObjId*(S))]
successor(*intV*(i)) = *intV*(i+1)
`initialState` = *state*(`emptyS`, `emptyS`[objIdLoc, *intV*(2)])

Figure 4.15: The definition of the abstract machine functions used in the operational and axiomatic semantics of Java-C.

Figure 4.7, maps a *TypeId* to its type declaration information (*TypeId* → *TypeDecl*). In our computational model, `TEnv` is the only component specific to a particular program.

To simplify our semantic rules, we also assume that *expr* and *stmt* are parse trees decorated with information from type checking and name resolution. For example, *typeOf* `(super)` yields the superclass of the current type declaration; this information is taken from the annotated parse tree. We also assume that the *TypeId* of the enclosing class is always available through the expression *typeOf* `(this)`.

4.5.1 Operational Semantic Rules of Java-C

Figures 4.16 through 4.19 give the operational semantics of Java-C. Figure 4.16 gives the semantics of expression evaluation and Figure 4.17 gives the semantics of the usual programming language statements; these are standard and require no further explanation except for the return statement. The return statement does not exit the method (it only sets the return value), so we assume

(L-Self) $[\texttt{this}, \mathsf{S}] \Rightarrow_{\text{lv}} \texttt{thisLoc}$

(L-VarId) $[x, \mathsf{S}] \Rightarrow_{\text{lv}} \mathit{loc}(x, \texttt{local})$

(L-Field) $$\frac{[vr, \mathsf{S}] \Rightarrow_{\text{e}} \text{r}}{[vr\,.\,x, \mathsf{S}] \Rightarrow_{\text{lv}} \mathit{loc}(x,\ \text{r})}$$

(L-Result) $[\texttt{\textbackslash result}, \mathsf{S}] \Rightarrow_{\text{lv}} \texttt{resultLoc}$

(E-VarRef) $$\frac{[vr, \mathsf{S}] \Rightarrow_{\text{lv}} \text{vLoc}}{[vr, \mathsf{S}] \Rightarrow_{\text{e}} \mathit{getValue}(\mathsf{S}, \text{vLoc})}$$

(E-Literal) $[lit, \mathsf{S}] \Rightarrow_{\text{e}} \mathit{mkVal}(lit)$

(E-BinOp) $$\frac{[e1, \mathsf{S}] \Rightarrow_{\text{e}} \text{v1},\ [e2, \mathsf{S}] \Rightarrow_{\text{e}} \text{v2}}{[e1\ bop\ e2, \mathsf{S}] \Rightarrow_{\text{e}} \mathit{apply}(bop, \text{v1}, \text{v2})}$$

(E-UnOp) $$\frac{[e, \mathsf{S}] \Rightarrow_{\text{e}} \text{v}}{[uop\ e, \mathsf{S}] \Rightarrow_{\text{e}} \mathit{apply}(uop, \text{v})}$$

(E-Paren) $$\frac{[e, \mathsf{S}] \Rightarrow_{\text{e}} \text{v}}{[\,\texttt{(}e\texttt{)}\,, \mathsf{S}] \Rightarrow_{\text{e}} \text{v}}$$

(E-CastNull) $[\,\texttt{(T)null}, \mathsf{S}] \Rightarrow_{\text{e}} \mathit{voidV}(\text{null})$

(E-Cast) $$\frac{[e, \mathsf{S}] \Rightarrow_{\text{e}} \text{r},\ \text{r} \in \mathit{ObjectRef},\ \mathit{refType}(\text{r}) \leq \text{T}}{[\,\texttt{(T)}\,e, \mathsf{S}] \Rightarrow_{\text{e}} \text{r}} \qquad \textbf{if } !(e \equiv \texttt{null})$$

Figure 4.16: Operational semantics of L-expressions and R-expressions in Java-C.

(S-Skip) $\{\ ;\ , \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}$

(S-IfThen)
$$\frac{[e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathit{boolV}(\mathrm{true}),\ \{C1, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'}{\{\texttt{if (}e\texttt{)}\ C1\ \texttt{else}\ C2, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'}$$

(S-IfElse)
$$\frac{[e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathit{boolV}(\mathrm{false}),\ \{C2, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'}{\{\texttt{if (}e\texttt{)}\ C1\ \texttt{else}\ C2, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'}$$

(S-While)
$$\frac{[e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathit{boolV}(\mathrm{true}),\ \{C\ \texttt{while (}e\texttt{)}\ C, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'}{\{\texttt{while (}e\texttt{)}\ C, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'}$$

(S-EndWhile)
$$\frac{[e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathit{boolV}(\mathrm{false})}{\{\texttt{while (}e\texttt{)}\ C, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}}$$

(S-Seq)
$$\frac{\{C1, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}',\ \{C2, \mathsf{S}'\} \Rightarrow_{\mathrm{s}} \mathsf{S}''}{\{C1\ \ C2, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}''}$$

Figure 4.17: Operational semantics of statements that do not directly involve method calls or assignments.

that the type checker ensures that the return statement can only occur as the last statement of a method and that every method that returns a value must have such a return statement.

Figure 4.18 provides the semantics of method calls. We provide rules to handle each kind of call, i.e., self-call (S-Call), super-call (S-SupCall and S-SupConstr), and object-call (S-Call). New object constructor calls are handled by the S-NewAssign rules of Figure 4.19. Since expressions do not have side-effects, the result returned from a method call has to be assigned to a local variable when the result needs to be used in a later expression; Figure 4.19 gives the rules for assignment statements.

The semantics of method calls must also handle dynamic binding. *Dynamic binding* is the run-time process of binding a method invocation to the code of a specific implementation, i.e., the code that is executed is determined based on the run-time type of the receiver object, not its static type. Dynamic binding presents a challenge because the code that is executed is determined at run-time, not statically.

$$\frac{[e0, \mathsf{S}] \Rightarrow_e \mathrm{r},\ [e, \mathsf{S}] \Rightarrow_e \mathrm{v},\ \mathrm{T} = \mathit{whereMethodDecl}(\ \mathit{refType}(\mathrm{r}),\ m\ ),\ \mathsf{S}' = \mathit{state}(\mathtt{emptyS}, \mathit{heapOf}(\mathsf{S}))[\mathtt{thisLoc} := \mathrm{r}][\mathit{loc}(\mathtt{p}, \mathtt{local}) := \mathrm{v}],\ \{\mathit{getBody}(\mathrm{T}, m), \mathsf{S}'\} \Rightarrow_s \mathsf{S}''}{\{e0\,.\,m\,(e)\,, \mathsf{S}\} \Rightarrow_s \mathit{state}(\mathit{stackOf}(\mathsf{S}), \mathit{heapOf}(\mathsf{S}''))} \quad \text{(S-Call)}$$

$$\frac{\mathrm{T} = \mathit{whereMethodDecl}(\ \mathit{superOf}(\mathit{typeOf}(\mathtt{this})),\ m\ ),\ [\mathtt{this}, \mathsf{S}] \Rightarrow_e \mathrm{r},\ [e, \mathsf{S}] \Rightarrow_e \mathrm{v},\ \mathsf{S}' = \mathit{state}(\mathtt{emptyS}, \mathit{heapOf}(\mathsf{S}))[\mathtt{thisLoc} := \mathrm{r}][\mathit{loc}(\mathtt{p}, \mathtt{local}) := \mathrm{v}],\ \{\mathit{getBody}(\mathrm{T}, m), \mathsf{S}'\} \Rightarrow_s \mathsf{S}''}{\{\mathtt{super}\,.\,m\,(e)\,, \mathsf{S}\} \Rightarrow_s \mathit{state}(\mathit{stackOf}(\mathsf{S}), \mathit{heapOf}(\mathsf{S}''))} \quad \text{(S-SupCall)}$$

$$\frac{[e, \mathsf{S}] \Rightarrow_e \mathrm{v},\ [\mathtt{this}, \mathsf{S}] \Rightarrow_e \mathrm{r},\ \mathrm{T} = \mathit{typeOf}(\mathtt{super}),\ \mathsf{S}' = \mathit{state}(\mathtt{emptyS}, \mathit{heapOf}(\mathsf{S}))[\mathtt{thisLoc} := \mathrm{r}][\mathit{loc}(\mathtt{p}, \mathtt{local}) := \mathrm{v}],\ \{\mathit{getBody}(\mathrm{T}, \mathrm{T}), \mathsf{S}'\} \Rightarrow_s \mathsf{S}''}{\{\mathtt{super}(e)\,, \mathsf{S}\} \Rightarrow_s \mathit{state}(\mathit{stackOf}(\mathsf{S}), \mathit{heapOf}(\mathsf{S}''))} \quad \text{(S-SupConstr)}$$

Figure 4.18: Operational semantics of method calls in Java-C.

Dynamic binding only applies to self-calls and object-calls since the method executed in super-calls and new object constructor calls is determined statically. Therefore, only the S-Call rule has to model dynamic binding. Dynamic binding is modeled in the S-Call rule by first evaluating the receiver expression *e0* and then extracting the run-time type from the resulting object reference; then the method is looked up in the type environment using the method name and receiver *e0*'s run-time type. The body of the method declaration obtained from this lookup is then executed (see function *getBody* of Figure 4.9). Note that, in our language subset, the receiver expression *e0* has to be `this` or a field or local variable name.

4.6 Axiomatic Semantics of JML-C

In this section we present a Hoare-style verification logic for Java-C; the axioms and inference rules are given in Figures 4.20 through 4.23. The *inference rules* have one or more assertions above the line and one below it. The assertions above the line are the *antecedents* or *premises* of the rule and the assertion below the line is its *consequent* or *conclusion*; the A-If rule of Figure 4.20 is an example. *Axioms* do not have antecedents, e.g., the A-LocalDecl, A-Return, A-ExpAssign rules of Figure 4.22.

$$\text{(S-ExpAssign)}\quad \frac{[e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathrm{v},\ [vr, \mathsf{S}] \Rightarrow_{\mathrm{lv}} \mathrm{vLoc}}{\{vr\texttt{=}e\texttt{;}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}[\mathrm{vLoc} := \mathrm{v}]}$$

$$\text{(S-LocalDecl)}\quad \frac{\mathrm{vLoc} = loc(\mathrm{x}, \texttt{local}),\ \mathrm{v}=mkVal(default(\mathrm{T}))}{\{\texttt{T x;}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}[\mathrm{vLoc} := \mathrm{v}]}$$

$$\text{(S-Return)}\quad \frac{[e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathrm{v}}{\{\texttt{return } e\texttt{;}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}[\texttt{resultLoc} := \mathrm{v}]}$$

$$\text{(S-CallAssign)}\quad \frac{\{e0\texttt{.}m\texttt{(}e\texttt{)}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}',\ [vr, \mathsf{S}] \Rightarrow_{\mathrm{lv}} \mathrm{vLoc},\ [\texttt{\textbackslash result}, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathrm{v}}{\{vr\texttt{=}e0\texttt{.}m\texttt{(}e\texttt{)}\texttt{;}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'[\mathrm{vLoc} := \mathrm{v}]}$$

$$\text{(S-SupCallAssign)}\quad \frac{\{\texttt{super.}m\texttt{(}e\texttt{)}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}',\ [vr, \mathsf{S}] \Rightarrow_{\mathrm{lv}} \mathrm{vLoc},\ [\texttt{\textbackslash result}, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathrm{v}}{\{vr\texttt{=super.}m\texttt{(}e\texttt{)}\texttt{;}, \mathsf{S}\} \Rightarrow_{\mathrm{s}} \mathsf{S}'[\mathrm{vLoc} := \mathrm{v}]}$$

$$\text{(S-NewAssign)}\quad \frac{\begin{array}{l}[vr, \mathsf{S}] \Rightarrow_{\mathrm{lv}} \mathrm{varLoc},\ [e, \mathsf{S}] \Rightarrow_{\mathrm{e}} \mathrm{v},\ new(\mathsf{S}, \mathrm{T}) = (\mathrm{r}, \mathsf{S}'),\\ \mathsf{S}'' = state(\texttt{emptyS}, heapOf(\mathsf{S}'))[\texttt{thisLoc} := \mathrm{r}][loc(\texttt{p}, \texttt{local}) := \mathrm{v}],\\ \{getBody(\mathrm{T}, \mathrm{T}), \mathsf{S}''\} \Rightarrow_{\mathrm{s}} \mathsf{S}'''\end{array}}{[vr\texttt{=new T(}e\texttt{)}\texttt{;}, \mathsf{S}] \Rightarrow_{\mathrm{s}} state(stackOf(\mathsf{S}), heapOf(\mathsf{S}'''))[\mathrm{varLoc} := \mathrm{r}]}$$

Figure 4.19: Operational semantics of assignment statements in Java-C.

Axioms assert a property that is assumed to always be true. The inference rules are used to prove new properties from the antecedents.

The purpose of our formal system is so verifiers can prove the correctness of a Java-C implementation with respect to its JML-C specification. The verification logic, together with our rules from Chapters 2 and 3 (as formalized in Section 5.1), must be sound for proving the correctness of subclass methods without superclass code, i.e., verifiers must be able to prove correctness of subclass methods using only the superclass and subclass specifications and our verification logic. If the rules from Chapters 2 and 3 have been checked statically by our tool (or some other similar tool), then the

(A-Skip) $\{P\}$ `;` $\{P\}$

(A-If)
$$\frac{\{P \texttt{ \&\& } e\}\ C1\ \{Q\},\ \{P \texttt{ \&\& } !e\}\ C2\ \{Q\}}{\{P\}\ \texttt{if (}e\texttt{)}\ C1\ \texttt{else}\ C2\ \{Q\}}$$

(A-While)
$$\frac{\{P \texttt{ \&\& } e\}\ C\ \{P\}}{\{P\}\ \texttt{while (}e\texttt{)}\ C\ \{P \texttt{ \&\& } !e\}}$$

(A-Seq)
$$\frac{\{P\}\ C1\ \{Q\},\ \{Q\}\ C2\ \{R\}}{\{P\}\ C1\ C2\ \{R\}}$$

(A-Conseq)
$$\frac{P \Rightarrow P',\ \{P'\}\ C\ \{Q'\},\ Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}$$

(A-ModelRep)
$$\frac{\{P\},\ \text{represents } \texttt{this.}F \texttt{ <- } e;}{\{P\}\ [\texttt{this.}F \leftarrow e]}$$

(A-ExpRep)
$$\frac{\{P\},\ \text{represents } \texttt{this.}F \texttt{ <- } e;}{\{P\}\ [e \leftarrow \texttt{this.}F]}$$

Figure 4.20: Axioms and rules for Java-C statements that do not directly involve assignment or method calls.

verifier can use our programming logic without the need to be concerned about downcalls or aliasing. In other words, the soundness of our verification logic depends on the enforcement of the rules given in Chapters 2 and 3 and formalized in Section 5.1.

4.6.1 Hoare Triples

Our verification logic builds on known techniques for specifying the axiomatic semantics of programming languages; in particular, the main formulae of the logic are Hoare triples [Hoa69]. A *Hoare triple* has the form {**P**} *S-Unit* {**Q**} where *S-Unit* denotes a syntactic unit. A *syntactic unit* (or *s-*

(A-SelfCall)
$$\frac{\begin{array}{l}U = typeOf(\texttt{this}),\ T = whereMethodDecl(U, m),\\ (\forall\ T1 \in TypeId\ :\\ \quad \{inv(T1)\ \texttt{\&\&}\ req(T1, m)\}\ getBody(T1, m)\ \{inv(T1)\ \texttt{\&\&}\ ens(T1, m)\}\ )\end{array}}{\{inv(U)\ \texttt{\&\&}\ req(T, m)[\texttt{p}\leftarrow e]\}\ \ \texttt{this.}m\texttt{(}e\texttt{)}\ \ \{inv(U)\ \texttt{\&\&}\ ens(T, m)\}}$$

(A-ObjCall)
$$\frac{\begin{array}{l}U = typeOf(\texttt{this}),\ T = whereMethodDecl(typeOf(vr), m),\\ (!(e \equiv \texttt{this})\ \vee\ inv(U)),\ !(vr \equiv \texttt{this}),\\ \{inv(T)\ \texttt{\&\&}\ req(T, m)\}\ \ getBody(T, m)\ \{inv(T)\ \texttt{\&\&}\ ens(T, m)\}\end{array}}{\{req(T, m)[\texttt{p}\leftarrow e, \texttt{this}\leftarrow vr]\}\ \ vr\texttt{.}m\texttt{(}e\texttt{);}\ \ \{ens(T, m)[\texttt{this}\leftarrow vr]\}}$$

(A-SupCall)
$$\frac{\begin{array}{l}U = typeOf(\texttt{this}),\ T = whereMethodDecl(superOf(U), m),\\ \{inv(T)\ \texttt{\&\&}\ req(T, m)\}\ \ getBody(T, m)\ \{inv(T)\ \texttt{\&\&}\ ens(T, m)\}\end{array}}{\{inv(U)\ \texttt{\&\&}\ req(T, m)[\texttt{p}\leftarrow e]\}\ \ \texttt{super.}m\texttt{(}e\texttt{)}\ \ \{inv(U)\ \texttt{\&\&}\ ens(T, m)\}}$$

(A-SupConstr)
$$\frac{\begin{array}{l}T = typeOf(\texttt{super}),\\ \{reqOf(T, m)\}\ \ getBody(T, T)\ \ \{inv(T)\ \texttt{\&\&}\ ensOf(T, m)\}\end{array}}{\{reqOf(T, m)[\texttt{p}\leftarrow e]\}\ \ \texttt{super(}e\texttt{);}\ \ \{inv(T)\ \texttt{\&\&}\ ensOf(T, m)\}}$$

Figure 4.21: Inference rules for method and constructor calls.

unit) is a program statement or sequence of statements. **P** and **Q** are assertions about the program state before and after execution of the given s-unit. In our deductive system, **P** and **Q** are JML-C predicates[8] that specify properties of the pre- and post-states of the given s-unit.

The triple {**P**} *S-Unit* {**Q**} asserts a partial correctness property of the given s-unit and has the following meaning: if **P** holds in the pre-state, then the execution of *S-Unit* either (1) terminates normally with assertion **Q** holding in the post-state, (2) abnormally terminates, or (3) runs forever. An s-unit may abnormally terminate because of program bugs or execution errors that are beyond the semantics of Java-C, e.g., stack overflow, running out of memory, null object references, etc. An s-unit, for example, runs forever if it executes in an infinite loop. However, our primary concern is with

8. These predicates are a subset of the Java-C expressions. JML also provides syntax for other logical constructs, such as quantifiers, but for simplicity we will not consider them here.

the properties of s-units when they terminate normally. The standard Hoare inference rules for procedural programming language statements are given in Figure 4.20 [Hoa69].

4.6.2 Method Verification

As described earlier, method behavior is specified in JML-C through requires, ensures, and assignable clauses[9]. However, to hide the concrete implementation, these clauses usually reference public model fields. To handle model fields in public pre- and post conditions and public invariants, references to a model field are replaced by the right hand side of that model field's represents clause. A represents clause specifies an abstraction function that maps the concrete state of the receiver object to the abstract value of the model field, i.e., it defines the concrete representation of that model field. For simplicity, we assume that all represents clauses specify an abstraction function with a consistent, well defined meaning [LM06], e.g., there are no cyclic or mutually recursive constraints among multiple model fields (see assumptions in subsection 1.6.6). The Java-C implementation of a method must be correct with respect to these JML-C specifications.

The A-SelfCall, A-SupCall, A-ObjCall, and A-SupConstr rules of Figure 4.21 specify how to reason about method and constructor invocations; these rules depend on the correctness proofs of their method or constructor implementations. That is, an antecedent in each of these rules specifies that the body of the method or constructor must satisfy its specification, e.g., {**P**} *getBody*(T, *m*) {**Q**} when method T**.***m* is being called.

These antecedents involving function *getBody* in the rules given in Figure 4.21 define what is meant for a method or constructor to be correct, i.e., how to verify the correctness of a method or constructor implementation with respect to its JML-C specification. Notice that the invariant of the enclosing class is a conjunct in both the precondition and postcondition of method calls, e.g., *inv*(T) must hold in the pre-state and it must also be satisfied upon exit from a method declared in class T. The precondition is extracted from the specification in the current class and combined with the inherited specifications from its superclasses; this ensures that implementations of subclass methods satisfy the combined superclass and subclass specifications.

Verifying the correctness of a constructor is similar; however, the class invariant is not a precondition for constructors because the object has not yet been initialized. Also, constructors cannot inherit specifications from superclasses, i.e., a subclass constructor cannot override a superclass constructor; also, *reqOf*(T, T) = *req*(T, T) and *ensOf*(T, T) = *ens*(T, T) for any class T.

4.6.3 Dynamic Binding

To handle dynamic binding, our verification logic requires that overriding methods inherit and satisfy the superclass specification as well as the subclass specification, i.e., a subclass must be a

9. The callable clause is not allowed to reference model fields; the receiver must be `this`, `super`, a formal parameter, or the static type of the receiver of an object-call (see subsection 5.1.3.1).

(A-LocalDecl) $\{P[\texttt{x} \leftarrow \mathit{default}(\texttt{T})]\}\ \texttt{T x;}\ \{P\}$

(A-Return) $\{P[\texttt{\result} \leftarrow e]\}\ \texttt{return}\ e\texttt{;}\ \{P\}$

(A-ExpAssign) $\{P[\texttt{vr} \leftarrow e]\}\ \mathit{vr} = e\texttt{;}\ \{P\}$

(A-CallAssign)
$$\frac{\{P\}\ e0\texttt{.m(}e\texttt{)}\ \{Q\}}{\{P\}\ \mathit{vr}\texttt{=}e0\texttt{.m(}e\texttt{);}\ \{Q[\texttt{\result}\leftarrow \mathit{vr}]\}}$$

(A-SupCallAssign)
$$\frac{\{P\}\ \texttt{super.m(}e\texttt{);}\ \{Q\}}{\{P\}\ \mathit{vr}\texttt{=super.m(}e\texttt{);}\ \{Q[\texttt{\result}\leftarrow \mathit{vr}]\}}$$

(A-NewAssign)
$$\frac{\{\mathit{reqOf}(\mathrm{T}, m)\}\ \texttt{T.T}\ \{\mathit{inv}(\mathrm{T})\ \texttt{\&\&}\ \mathit{ensOf}(\mathrm{T}, m)\},\ (!(e \equiv \texttt{this}) \vee \mathit{inv}(\mathit{typeOf}(\texttt{this})))}{\{\mathit{reqOf}(\mathrm{T}, m)[\texttt{p}\leftarrow e]\}\ \mathit{vr}\texttt{=new T(}e\texttt{);}\ \{(\mathit{inv}(\mathrm{T})\ \texttt{\&\&}\ \mathit{ensOf}(\mathrm{T}, m))[\texttt{this}\leftarrow \mathit{vr}]\}}$$

Figure 4.22: Axioms and inference rules for statements that modify the program state.

behavioral subtype of its superclasses. The functions *req*, *ens*, and *assigns* defined in Figure 4.11 make sure the superclass specification is inherited. Method correctness requires that a method implementation satisfy its inherited requires and ensures clauses since it uses the functions *req* and *ens* when determining the pre- and postconditions for a method. The function *req* forms the precondition by disjoining (or-ing) the precondition from a method declaration with the preconditions of each of the superclass methods it overrides. Similarly, function *ens* forms the postcondition by conjoining the postcondition from a method declaration with the postconditions of the methods it overrides. However, to achieve the intended meaning, the postcondition of each specification case has to be changed slightly, i.e., the postcondition in each specification case must be implied by the corresponding precondition when evaluated in the pre-state, e.g., (!`\old`(**P**) || **Q**) must be the postcondition for each case; this is what the function *ens* does as it traverses the class hierarchy (see function *postCond* of Figure 4.11).

The clauses occurring in separate specification cases of the same method declaration are also desugared in the same way, i.e., by disjoining the preconditions and conjoining the postconditions as

$$\text{(A-OldVerify)}\quad \frac{\{\mathrm{P}\ \texttt{\&\&}\ Z\texttt{==}e\}\ \ C\ \ \{\mathrm{Q}[\texttt{\textbackslash old}(e)\leftarrow Z]\}}{\{\mathrm{P}\}\ \ C\ \ \{\mathrm{Q}\}} \quad \textbf{if } Z \in \mathit{LogicId}$$

$$\text{(A-OldCall)}\quad \frac{\{\mathrm{P}\}\ \ C\ \ \{\mathrm{Q}\}}{\{\mathrm{P}\ \texttt{\&\&}\ Z\texttt{==}e\}\ \ C\ \ \{\mathrm{Q}[\texttt{\textbackslash old}(e)\leftarrow Z]\}} \quad \textbf{if } Z \in \mathit{LogicId} \text{ and } C \text{ is a method call}$$

$$\text{(A-SpecCase)}\quad \frac{\{\mathrm{P}\}\ C\ \{\mathrm{Q}\},\ \{\mathrm{P}'\}\ C\ \{\mathrm{Q}'\}}{\begin{array}{l}\{\ (\mathrm{P}\ \|\ \mathrm{P}')\ \texttt{\&\&}\ Z\texttt{==}\mathrm{P}\ \texttt{\&\&}\ Z'\ \texttt{==}\mathrm{P}'\}\\ \quad C\\ \{(!Z\ \|\ \mathrm{Q})\ \ \texttt{\&\&}\ \ (!Z'\ \|\ \mathrm{Q}')\}\end{array}} \quad \textbf{if } Z, Z' \in \mathit{LogicId}$$

$$\text{(A-Assignable)}\quad \frac{\begin{array}{l}\mathrm{T} = \mathit{typeOf}(e0),\ \ w \notin_{\mathrm{a}} \mathit{selfAssigns}(e0, \mathrm{T}, \texttt{m}),\\ w \notin_{\mathrm{a}} \mathit{parmAssigns}(e, \mathrm{T}, \texttt{m}),\ \ \{\mathrm{P}\}\ \ e0\texttt{.m(}e\texttt{)}\ \ \{\mathrm{Q}\}\end{array}}{\{\mathrm{P}\ \texttt{\&\&}\ Z\texttt{==}w\}\ \ e0\texttt{.m(}e\texttt{)}\ \ \{\mathrm{Q}\ \texttt{\&\&}\ Z\texttt{==}w\}} \quad \textbf{if } Z \in \mathit{LogicId}$$

$$\text{(A-SupAssignable)}\quad \frac{\begin{array}{l}\mathrm{U} = \mathit{typeOf}(\texttt{this}),\ \ \mathrm{T} = \mathit{whereMethodDecl}(\mathit{superOf}(\mathrm{U}), m),\\ \mathrm{U} \le \mathrm{T1} < \mathrm{T},\ f \in \mathit{setOfFieldsIn}(\mathrm{T1}),\\ (w \equiv \texttt{this}.f\ \vee\ w \equiv \texttt{this}.f.g),\\ \{\mathrm{P}\}\ \ \texttt{super.m(}e\texttt{)}\ \ \{\mathrm{Q}\}\end{array}}{\{\mathrm{P}\ \texttt{\&\&}\ Z\texttt{==}w\}\ \ \texttt{super.m(}e\texttt{)}\ \ \{\mathrm{Q}\ \texttt{\&\&}\ Z\texttt{==}w\}} \quad \textbf{if } Z \in \mathit{LogicId}$$

Figure 4.23: Inference rules involving logical variables.

described above; we also assume that the specification cases for a specific method declaration have already been desugared during type checking as the type environment is being created. Therefore, *ensOf* and *reqOf* return the combined, desugared specification cases for a specific method declaration, whereas *ens* and *req* combine the inherited specification cases with the subclass specification.

4.6.4 Object and Class Invariants

We say that an object is in a *consistent state* if its run-time type invariant holds. The A-SelfCall, A-SupCall, and A-ObjCall rules of Figure 4.21 require that the receiver `this` be in a consistent state

whenever it is passed as an argument. For example, *inv*(U) is a conjunct in the pre- and postconditions of the A-SelfCall and A-SupCall rules. Similarly, *inv*(U) must hold in the pre-state whenever `this` is the non-receiver parameter in an object-call, i.e., one the antecedents of the A-ObjCall rule requires that *inv*(U) hold prior to the call when $e \equiv$ `this`. Note also that the same antecedent appears in the A-NewAssign rule of Figure 4.22 for the same reason, i.e., to ensure that the current receiver is in a consistent state before it can be passed as an argument in a new object constructor call. In summary, the invariant of the current receiver can temporarily be invalid as long as it holds prior to being passed as an argument in a call or prior to exiting the method or constructor.

Furthermore, in the pre-state, method calls expect all of their argument objects to be in a consistent state [MPHL05]. Therefore, the receiver of an object-call must also be in a consistent state. Hence, the A-ObjCall rule implicitly assumes that the run-time type invariant of the receiver *vr* holds in the pre-state (note that this will have to be proved in Chapter 5).

4.6.5 Assignment Statements

The rules given in Figure 4.22 are for statements that directly change the program state. These rules extend the standard assignment axiom (i.e., A-ExpAssign). These rules handle local declarations, return statements, and method or constructor calls that assign their return value to a variable. In a postcondition, the JML keyword \result denotes the value returned from the method call (see the A-CallAssign and A-SupCallAssign rules of Figure 4.22).

4.6.6 Logical Variables

The only free variables allowed in pre- and postconditions are *logical variables* (*LogicId*). *Free variables* are those not bound in the type environment `TEnv` (see Figure 4.23). Therefore, the domains *LogicId* and *VarId* are disjoint; this also prevents name conflicts. These logical variables are implicitly quantified at the outermost logical level.

The rules given in Figure 4.23 present rules for handling special situations and properties of our verification logic. For example, the A-OldVerify rule specifies how a verifier can remove expression `\old`(*e*) from a method's postcondition before proving its correctness. The semantics of expression `\old`(*e*) requires that *e* be evaluated in the pre-state; thus, to handle such situations, the semantics would have required two states, the pre- and post-states, when determining whether or not a given postcondition holds. The purpose of this rule is to eliminate the need for the pre-state when evaluating postconditions, i.e., only the post-state is needed.

The A-OldCall rule is for use when removing `\old`(*e*) from the postcondition of a called method; the precondition fixes the value of logical variable *Z* to the value of *e* evaluated in the pre-state; then `\old`(*e*) is replaced by *Z* in the postcondition; thus *Z* denotes the same pre-state value in the pre- and postconditions. The soundness of the A-OldCall and A-OldVerify rules will demonstrate the soundness of our handling of `\old`-expressions in postconditions.

Similarly, the soundness of the A-SpecCase rule will demonstrate the soundness of our desugaring of specification cases and, in particular, the way the functions *req* and *ens* of Figure 4.11 combine and desugar specification cases that are inherited from superclasses. This rule also demonstrates that our desugaring of specification cases satisfies the intended meaning of specification cases. Specifically, if the method body satisfies each specification case separately, then the consequent of this rule should be a single Hoare triple with the combined, desugared pre- and postconditions that are derived from the two specification cases. Note that the consequent of the A-SpecCase rule is derived from the desugaring of the two specification cases (Hoare triples) above the line, followed by two applications of the A-OldCall rule (i.e., the A-OldCall rule introduces the two logical variables, Z and Z' and removes the two `\old`-expressions from the postcondition that were introduced by the desugaring, e.g., by function *ens*). In summary, the soundness of the A-SpecCase rule demonstrates the soundness of our desugaring of specification cases and it allows the verifier to prove correctness of each specification case separately (see the examples in Section 4.7).

We have also included the two rules A-Assignable and A-SupAssignable because they verify that no unexpected changes in the state of objects is possible. Therefore, only those concrete and model fields permitted by the assignable clauses can change during a method call. That is, the A-Assignable and A-SupAssignable rules specify that only members of the set of assignable model and concrete fields are allowed to change; thus, variables that are not assignable during a method call will be the same before and after that call. In addition, the A-SupAssignable rule says that no subclass field changes during a super-call. The soundness of these rules demonstrate the soundness of the Additional Side-Effects Invalidation Rule from Chapter 2 (no subclass fields change) and the soundness of our alias control technique from Chapter 3 (only fields permitted by the assignable clause can change).

4.7 Example Correctness Proofs

In this section, we give an example of a correctness proof for several methods and classes in a class hierarchy. We prove that these methods are correct with respect to their specifications to illustrate how our verification logic would be used by verifiers.

For simplicity, the Java-C grammar requires that each method have exactly one parameter. Thus to handle this situation, we consider a method with no parameters to be syntactic sugar for a method with one formal parameter with type `int`; furthermore, the method body and its specification cannot and do not reference that formal parameter. Similarly, the call of a method with no parameters implicitly passes 0 as the corresponding actual parameter.

Also, in JML, when the formal parameter is referenced in the post condition it is implicitly enclosed in a `\old`-expression since the formal parameter may have been assigned to during the execution of that method, i.e., we do not allow the postcondition to reference stack variables or formal parameters in the post-state. However, to avoid any ambiguity in the examples below, we have

```
public class IntValue {
   //@ public model int value;      // model variable

  /*@  public normal_behavior
    @     assignable this.value;
    @     ensures this.value == \old(initVal);   @*/
  public IntValue(int initVal);

  /*@  public normal_behavior
    @     ensures \result == this.value;   @*/
  public /*@ pure @*/ int get();

  /*@  public normal_behavior
    @     assignable this.value;
    @     ensures this.value == \old(newVal);  @*/
   public void set(int newVal);

  /*@  public normal_behavior
    @     requires v != null;
    @     assignable this.value;
    @     ensures this.value == \old(v.value)
    @         && \result == this.value - \old(this.value);  @*/
   public int setFrom(IntValue v);


  protected int _val;
  //@              in value;

  //@ protected represents value <- _val;
}
```

Figure 4.24: `IntValue`'s combined public and protected specification from file `IntValue.jml-refined`.

explicitly included the `\old()` in these postconditions (see methods `set` and `setFrom` of Figure 4.24).

Consider first the combined public and protected specification of class `IntValue` given in Figure 4.24. We now prove the correctness of method `setFrom` of Figure 4.25; the correctness proofs of the other methods in class `IntValue` are trivial and are left to the reader. In the proof, the A-Seq and A-Conseq rules of Figure 4.20 will be used implicitly at each step; the other rules will be explicitly cited when they are used. The proofs will be presented with intermittent assertions in the annotation style of Hesselink [Hes92]; this is equivalent to a natural deduction style proof.

Since the postcondition of `setFrom` has two `\old`-expressions, we will introduce two logical variables *Z1* and *Z2* and will use the A-OldVerify rule of Figure 4.23 to verify correctness of the implementation. The first conjunct is the class invariant, i.e., true is the default invariant.

{true && `v` != `null` && *Z1* == `v.value` && *Z2* == `this.value`}
⇔ < logic and the semantics of && >
{`v` != `null` && *Z1* == `v.value` && *Z2* == `this.value`}

```
int vVal;
```

< by the A-LocalDecl axiom of Figure 4.22 >
{`v` != `null` && *Z1* == `v.value` && *Z2* == `this.value`}

```
int res;
```

< by the A-LocalDecl axiom of Figure 4.22 >
{`v` != `null` && *Z1* == `v.value` && *Z2* == `this.value`}

```
vVal = v.get();
```

< `v`, `v.value`, and `this.value` do not change (by the A-Assignable rule of Figure 4.23);
thus by the A-ObjCall rule of Figure 4.21 and the A-CallAssign rule of Figure 4.22 >
{`v` != `null` && *Z1* == `v.value` && *Z2* == `this.value`}
∧ {\result == `this.value`}[`this`←v, \result←vVal]
⇒ < meaning of substitution >
{`v` != `null` && *Z1* == `v.value` && *Z2* == `this.value`} ∧ {vVal == `v.value`}
⇒ < substitution since `vVal` = `v.value` >
{`v` != `null` && *Z1* == `vVal` && *Z2* == `this.value`}
⇒ < logic and the semantics of && >
{*Z1* == `vVal` && *Z2* == `this.value`}
⇒ < logic and the semantics of == and && >
{`vVal` - `this.value` == *Z1* − *Z2* && *Z1* == `vVal`}
⇒ < by the represents clause of `this.value` and the A-ModelRep rule of Figure 4.20 >
{`vVal` − `this._val` == *Z1* − *Z2* && *Z1* == `vVal`}

```
res = vVal - this._val;
```

< by the A-ExpAssign rule of Figure 4.22 >
{`res` == *Z1* − *Z2* && *Z1* == `vVal`}

```
if (this._val != vVal) {
```

{`this._val` != `vVal` && `res` == *Z1* − *Z2* && *Z1* == `vVal`}
⇒ < logic and the semantics of && >
{`res` == *Z1* − *Z2* && *Z1* == `vVal`}

```
this.set(vVal);
```

< `res` and `vVal` do not change (by the A-Assignable rule of Figure 4.23);
thus by the A-SelfCall rule of Figure 4.21 and the A-OldCall rule of Figure 4.23 >

{res == *Z1* − *Z2* && *Z1* == vVal} ∧ {this.value == *Z1*}
⇒ < logic and the semantics of == and && (substituting this.value for *Z1* in the first conjunct) >
{this.value == *Z1* && res == this.value − *Z2*}
} else {
{!(this._val != vVal) && res == *Z1* − *Z2* && vVal == *Z1*}
⇒ < logic and the semantics of ! and != >
{this._val == vVal && res == *Z1* − *Z2* && vVal == *Z1*}
⇒ < logic and the semantics of == and && (substituting *Z1* for vVal in the first conjunct and this._val for *Z1* in the second conjunct) >
{this._val == *Z1* && res == this._val − *Z2*}
⇒ < by the represents clause of this.value and the A-ExpRep rule of Figure 4.20 >
{this.value == *Z1* && res == this.value - *Z2*}
;
< by the A-Skip rule of Figure 4.20 >
{this.value == *Z1* && res == this.value - *Z2*}
}
< by the A-If rule of Figure 4.20 >
{this.value == *Z1* && res == this.value - *Z2*}
return res;
< by the A-Return axiom of Figure 4.22 >
{this.value == *Z1* && \result == this.value - *Z2*}
⇔ < logic and the semantics of && (to establish the invariant) >
{true && this.value == *Z1* && \result == this.value - *Z2*}

By the A-OldVerify rule, method setFrom satisfies its specification.

■

Now consider the public and protected specification of method set in subclass IntValuePlus given in Figure 4.26. This subclass keeps track of the previous value; it also has an invariant that constrains this._diff through the current and previous values. The subclass implementation of set is given in Figure 4.27 and its corresponding subclassing contract (to be used in another example below) is given in Figure 4.28. We now prove the correctness of the set method as implemented in Figure 4.27.

In the subclass IntValuePlus, method set has two specification cases, one inherited from the superclass and the other declared in the subclass IntValuePlus. Since the meaning of the combined specification cases is the conclusion of the A-SpecCase rule of Figure 4.23, it suffices to prove each specification case separately.

```
//@ refines "IntValue.jml-refined";

public class IntValue {
  protected int _val;

  public IntValue(int initVal) {
      _val = initVal;
  }
  public void get() {
      return _val;
  }
  public void set(int newVal) {
      _val = newVal;
  }
  public int setFrom(IntValue v) {
      int vVal;
      int res;
      vVal = v.get();
      res = vVal - this._val;
      if (this._val != vVal) {
          this.set(vVal);           // possible downcall here
      } else {
          ;
      }
      return res;
  }
}
```

Figure 4.25: `IntValue`'s implementation from the file `IntValue.java`.

Case 1: (from the superclass)

The postcondition of the superclass specification case has one `\old`-expression, so we will introduce one logical variable Z and will use the A-OldVerify rule of Figure 4.23 to verify correctness. The first conjunct in the first assertion is the subclass invariant and the second conjunct is the precondition from the superclass specification. (Note that we cannot use the superclass invariant in this proof even though the specification case comes from the superclass because the method has to establish the subclass invariant in the post-state; also, we are allowed to assume that the subclass invariant holds in the pre-state since the proof is for a method declared in the subclass.)

$\{$`this._diff == this.`value - `this.`oldVal && true && $Z ==$ `newVal`$\}$

$\Leftrightarrow <$ logic and the semantics of && $>$

$\{Z ==$ `newVal`$\}$

```
public class IntValuePlus extends IntValue {

  //@ public model int oldVal;
  //@                    in value;

    //  ...

  /*@ also
    @  public normal_behavior
    @    assignable value, oldVal;
    @    ensures oldVal == \old(value);   @*/
  public int set(int newVal);


    protected int _prevValue;
    //@                  in oldVal;
    //@ protected represents oldVal <- _prevValue;

    protected int _diff;
    //@            in value, oldVal;
    //@ protected invariant _diff == (value - oldVal);
}
```

Figure 4.26: Fragment of the specification of `IntValuePlus` from file `IntValuePlus.jml-refined`.

```
public class IntValuePlus extends IntValue {

  protected int _prevValue;
  protected int _diff;

  // ...

  public void set(int newVal) {
      this._prevValue = this._val;
      this._val = newVal;
      this._diff = this._val - this._prevValue;
  }
}
```

Figure 4.27: An implementation of method `set` in class Int`ValuePlus` from file Int`ValuePlus.java`.

```
this._prevValue = this._val;
```
< by the A-ExpAssign axiom of Figure 4.22 >
{*Z* == `newVal`}
```
this._val = newVal;
```
< by the A-ExpAssign axiom of Figure 4.22 >
{*Z* == `this._val`}
⇒ < logic and the semantics of == and && >
{*Z* == `this._val`
&& (`this._val - this._prevValue == this._val - this._prevValue`)}
```
this._diff = this._val - this._prevValue;
```
< by the A-ExpAssign axiom of Figure 4.22 >
{*Z* == `this._val && this._diff == this._val - this._prevValue`}
⇒ < logic and the represents clauses of `this.value` and `this.oldVal`
and the A-ExpRep rule of Figure 4.20 >
{`this._diff == this.value - this.oldVal` && *Z* == `this.value`}

Case 2: (from the subclass)

The postcondition of the subclass specification case has one `\old`-expression, so we will introduce one logical variable *Z* and will use the A-OldVerify rule of Figure 4.23 to verify correctness. As in Case 1, the first conjunct is the class invariant and the second conjunct is the precondition.

{`this._diff == this.value - this.oldVal` && true && *Z* == `this.value`}
⇒ < logic and the semantics of == and && >
{*Z* == `this.value`}
⇒ < by the represents clause of `this.value` and the A-ModelRep rule of Figure 4.20 >
{*Z* == `this._val`}
```
this._prevValue = this._val;
```
< by the A-ExpAssign axiom of Figure 4.22 >
{*Z* == `this._prevValue`}
```
this._val = newVal;
```
< by the A-ExpAssign axiom of Figure 4.22 >
{*Z* == `this._prevValue`}
⇒ < logic and the semantics of == and && >
{*Z* == `this._prevValue`
&& (`this._val - this._prevValue == this._val - this._prevValue`)}
```
this._diff = this._val - this._prevValue;
```
{*Z* == `this._prevValue && this._diff == this._val - this._prevValue`}
⇒ < logic and the represents clauses of `this.value` and `this.oldVal`
and the A-ExpRep rule of Figure 4.20 >

```
//@ refines "IntValuePlus.java";

public class IntValuePlus extends IntValue {

  // ...

  /*@ also
    @ protected code normal_behavior
    @    requires \same;
    @    callable \nothing;          @*/
  public int set(int newVal);

}
```

Figure 4.28: Part of `IntValuePlus`'s subclassing contract from file `IntValuePlus.refines-java`.

{`this._diff == this.value - this.oldVal && Z == this.oldVal`}

By the above two cases and the A-OldVerify and A-SpecCase rules of Figure 4.23, method `set` satisfies its specification.

■

We now give another example proof to illustrate the use of the A-SupCall rule of Figure 4.21 and the A-SupAssignable rule of Figure 4.23. Consider the specification of method `set` in the subclass `ValueTotal` given in Figure 4.29; `set`'s implementation is given in Figure 4.30. The super-call of `super.set` is allowed in this implementation because the superclass method does not make downcalls to methods with additional side-effects and it does not invalidate the subclass invariant (which, in this subclass, is true by default, and thus cannot be invalidated).

By the A-Conseq rule of Figure 4.20, when the preconditions of several specification cases are the same, we can prove that these specification cases are all satisfied in one proof, i.e., by starting from their common precondition and proving that the method's post-state must satisfy the conjunction of the postconditions from these specification cases (this conjunction implies the postcondition of each individual specification case). For example, the precondition of the two inherited specification cases and the new specification case is the same for method `set`, i.e., the precondition is true for all three cases. Therefore, we can prove that this method satisfies its specification by proving that if the common precondition holds in the pre-state, then the conjunction of the three postconditions must hold in the post-state; this is the approach we will take in our next proof.

Since each of the three postconditions of the specification cases of method `set` has one `\old`-expression, we will introduce three logical variables *Z1*, *Z2*, and *Z3* and will use the A-OldVerify rule

```
public class ValueTotal extends IntValuePlus {

  //@ public model int totalChg;
  //@                      in value;

  // ...

  /*@ also
    @  public normal_behavior
    @    assignable this.value, this.totalChg;
    @    ensures this.totalChg
    @            == \old(this.totalChg) + (this.value - this.oldVal);   @*/
  public void set(int newVal);


  protected int _total;
  //@               in totalChg;

  //@ protected represents totalChg <- _total;

}
```

Figure 4.29: A fragment of `ValueTotal`'s specification from the file `ValueTotal.jml-refined`.

```
//@ refines "ValueTotal.jml-refined";

public class ValueTotal extends IntValuePlus {
  protected int _total;

  // ...

  public int set(int newVal) {
      super.set(newVal);
      this._total = this._total + this._diff;
  }
}
```

Figure 4.30: A fragment of `ValueTotal`'s implementation from the file `ValueTotal.java`.

of Figure 4.23 to verify correctness. As in previous examples, the proof must include the type invariant.

{this._diff == this.value - this.oldVal && true
 && *Z1* == newVal && *Z2* == this.value && *Z3* == this.totalChg}
super.set(newVal);
  < newVal is unchanged (by the A-Assignable rule of Figure 4.23), and
    this.totalChg is unchanged (by the A-SupAssignable rule of Figure 4.23)
    even though super.set has permission to assign to this.totalChg since
    super.set does not make downcalls to methods with addition side-effects;
    by the A-SupCall rule of Figure 4.21 and the A-OldCall rule of Figure 4.23 >
{this._diff == this.value - this.oldVal && true
 && *Z1* == this.value && *Z2* == this.oldVal
 && *Z1* == newVal && *Z3* == this.totalChg}
 ⇒ < logic and the semantics of == and && and *Z3* == this.totalChg >
{this._diff == this.value - this.oldVal
 && *Z1* == this.value && *Z2* == this.oldVal
 && (this.totalChg + this._diff == *Z3* + this._diff)}
 ⇒ < by the represents clause of this.totalChg and the A-ModelRep rule of Figure 4.20 >
{this._diff == this.value - this.oldVal
 && *Z1* == this.value && *Z2* == this.oldVal
 && (this._total + this._diff == *Z3* + this._diff)}
this._total = this._total + this._diff;
{this._diff == this.value - this.oldVal
 && *Z1* == this.value && *Z2* == this.oldVal
 && this._total == *Z3* + this._diff}
 ⇒ < the semantics of == and substitution for this._diff in the last conjunct >
{this._diff == this.value - this.oldVal
 && *Z1* == this.value && *Z2* == this.oldVal
 && this._total == *Z3* + (this.value - this.oldVal)}
 ⇒ < by the represents clause of this.totalChg and the A-ExpRep rule of Figure 4.20 >
{this._diff == this.value - this.oldVal
 && *Z1* == this.value && *Z2* == this.oldVal
 && this.totalChg == *Z3* + (this.value - this.oldVal)}

The first conjunct of the last assertion above means the type invariant holds, the second conjunct says that the postcondition of method set of superclass IntValue holds, the third conjunct says that the postcondition in superclass IntValuePlus holds, and the fourth conjunct says that the

postcondition in subclass `ValueTotal` holds. Therefore, by the A-Conseq rule of Figure 4.20, the A-SpecCase rule of Figure 4.23, and the A-OldVerify rule of Figure 4.23, method `set` of class `ValueTotal` satisfies its specification.

■

4.8 Discussion

In this chapter, we specified the syntax and operational semantics of Java-C. We also specified the syntax and axiomatic semantics of JML-C. The operational semantics in Section 4.5 includes mechanisms that allow aliasing, code inheritance, and dynamic binding; this is necessary and important because the purpose of our technique is to prevent the problems caused by these features when creating subclasses. Therefore, the verification logic in Section 4.6 (together with the static enforcement of the rules from Chapters 2 and 3) must also handle aliasing, inheritance, and dynamic binding while preventing the associated problems.

The operational semantics of Java-C is an extension of the work of Arnd Poetzsch-Heffter and Peter Mueller. Our model of object storage is similar to the object environment in Poetzsch-Heffter's Habilitation Thesis [PH97] and the object store in Poetzsch-Heffter and Mueller's "*A Programming Logic for Sequential Java*" [PHM99]. However, our program state is represented as a pair of object stores (one for stack variables and the other for heap objects) whereas, in their work, the program state is a single object environment/object store containing both heap objects and local stack variables. The rest of the operational semantics of Java-C, although similar to theirs in some ways, was conceptually patterned after an interpreter for an object-oriented programming language that I created for a compiler class.

The syntax of Java-C is a slightly larger subset of Java than the one given in Poetzsch-Heffter and Mueller's paper. Like them, methods have only one formal parameter and expressions are not allowed to have side-effects, i.e., the result of a method call must be assigned to a variable and expressions with side-effects are not in the language. However, our language is different in that it more closely matches Java, including the syntax of super-calls, superclass constructor calls, and new object constructor calls since we are interested in the behavior of these kinds of calls. The languages are also different in that we have methods that do not return a value, i.e., methods with a void return type.

Our verification logic is an axiomatic semantics for the JML-C specification language. An important feature of JML-C is specification inheritance which forces subclasses to be behavioral subtypes [DL96]. In Poetzsch-Heffter and Mueller's paper subclasses also have to be behavioral subtypes, i.e., overriding subclass methods have to satisfy the specifications of overridden superclass methods. However, the rules in our verification logic are quite different because we use a type environment containing JML-C specifications and use functions like *inv*, *req*, *ens*, and *assigns* to extract and combine specification cases from superclasses. We also include rules, given in Figure 4.23, that are specific to JML-C such as the rules for eliminating `\old`-expressions, handling specification

cases, and reasoning about side-effects during method calls. We also have a rule, unique to our work, for reasoning about side-effects during super-calls, i.e., the A-SupAssignable rule in Figure 4.23.

In Chapter 5, we prove the soundness of our verification logic with respect to the operational semantics of Java-C. This soundness proof will verify whether or not our technique prevents the problems caused by downcalls and aliasing because our verification logic uses only superclass specifications, subclass specifications, and subclass code.

CHAPTER 5: SOUNDNESS OF OUR TECHNIQUE

In Chapter 4, we formally specified the syntax and semantics of a core subset of the Java programming language called Java-C. We also defined a verification logic for proving the correctness of Java-C programs with respect to their JML-C specifications. The verification logic is an axiomatic semantics for a core subset of the JML specification language.

In this chapter we prove that our technique successfully prevents the problems caused by downcalls and aliasing. That is, we prove that our verification logic is sound for proving the correctness of subclass methods without superclass code. For soundness, we need to prove that our verification logic is sound with respect to the operational semantics. However, our verification logic is unsound without an enforcement of the rules given in Chapters 2 and 3; thus these rules need to be formalized so they can be used in the soundness proof. These rules are formalized in Section 5.1 through a set of type rules; this formalization also demonstrates that the informal rules from Chapters 2 and 3 can be statically checked and enforced. Furthermore, when needed in our soundness proof, we can assume that programs satisfy the formal rules.

The main results of this chapter are presented in Section 5.2 where we prove that our verification logic is sound. Since we assume that superclass code is unavailable, we have to show that a superclass method can establish the run time type invariant prior to a call and specifically prior to a downcall. That is, we have to prove that a call of a superclass method (allowed by our technique) does not at any time during execution invalidate any of the subclass portions of the run-time type invariant; if a superclass method has this property, then the invariant of the run-time type of the receiver can be established by simply establishing the invariant of the static type of the receiver. The Valid Invariant Theorem 5.30 proves that during the execution of superclass methods (allowed by our checking rules), the run-time type invariant is established whenever the invariant of the static type of the receiver is established.

Our verification logic also requires that superclass methods make no assignments to subclass fields (through downcalls) since such side-effects are unverifiable without superclass code (as explained in subsection 2.2.3). The Additional Side-Effects Theorem 5.48 proves that during the call of a superclass method, allowed by our rules, subclass fields are not changed.

We also have to prove the soundness of our alias control technique since the soundness of our use of the assignable clause depends on preventing unexpected side-effects and unsafe aliasing. That is, we have to prove that the only fields that can change during the execution of a method are those permitted by its assignable clause. In the Assignable Clause Theorem 5.43, we prove that assignments to fields of the receiver and formal parameter are not permitted during method execution unless specified in its assignable clause. We also prove, in the Owner Aliasing Theorem 5.35, that when classes and methods satisfy our rules, verifiers can reason locally about aliasing and side-effects using owner variable names.

Finally, in Section 5.3, we conclude with a discussion of our formal rules and how some of them can be made less conservative and match more exactly with the informal rules given in Chapters 2 and 3. We used more restrictive rules in our formalization to help keep the soundness proof a little simpler and more manageable. Nonetheless, the soundness proof demonstrates that our technique works and can be used to avoid the problems caused by downcalls and aliasing.

5.1 Additional JML Type Checking

5.1.1 Overview

Our technique extends the type system of Java to include the rules given in Chapters 2 and 3; these rules simplify program verification, i.e., we extend Java's static type system so these additional rules do not have to be included in the operational semantics of Java-C or in the axiomatic semantics of JML-C specifications.

The additional type rules are called "T-rules" (short for type rules). In this subsection we give an overview of the purpose of these rules; in later subsections we explain more of the details and how they formalize the rules given in Chapters 2 and 3.

As explained in Chapter 2, one main goal of our technique is to eliminate the potential problems caused by downcalls. Downcalls can occur when a subclass method makes a super-call or when a subclass method makes a self-call to an unoverridden superclass method. In either case, a downcall occurs when the superclass method calls down to an overridden subclass method (see Figure 1.3). Since our technique assumes that the superclass code is not available, we have to prevent the execution of those superclass methods that have unverifiable side-effects or may no longer satisfy their specification. Therefore, our technique must disallow any such super-calls and it must make sure that the required methods have been overridden. Our technique uses the information in the assignable and callable clauses to determine when there are downcalls that invalidate a super-call, i.e., cause the super-call to be unsafe without superclass code. The T-rules given in Figures 5.1 - 5.3 formalize the Additional Side-Effects and Invariant Invalidation Rules given in Chapter 2.

The T-rules also formalize the alias control rules from Chapter 3 so customizers and verifiers can reason about the state of objects locally, i.e. modularly. Thus another main goal of our technique is to prevent objects in the current context from being changed unexpectedly through aliases visible in a different context. To accomplish this, our technique makes sure that all changes to the state of an object are initiated through an owner variable visible in the current context. Our technique must also make sure that changes to the state of an object cannot be initiated through a non-owner variable.

In summary, the T-rules (Figures 5.1 - 5.3) formalize the way most of the rules given in Chapters 2 and 3 can be statically enforced. Other rules are formalized by the functions defined in Figure 5.7 as explained in subsection 5.1.8, i.e., these functions specify how a static checker can ensure that the specification is well formed. For example, these functions formalize the checking of the data representation (i.e., represents, in, and maps clauses) to make sure fields have the proper data group

relationships (e.g. that the Pivot Declaration Rule of subsection 3.3.2 has been satisfied). These functions also check that the required methods have been overridden, i.e. that the overriding rules of Chapter 2 have been satisfied. Our technique checks the represents, in, maps, and assignable clauses to make sure that the specifications are well formed and that the required methods have been overridden.

Finally, the T-rules (Figures 5.1 - 5.3) also define how the assignable and callable clauses can be statically checked to ensure that the method implementation satisfies this part of the specification; subsection 5.1.7 describes how this checking would be done. Although the purpose of the T-rules is to ensure that methods satisfy the assignable and callable clauses, these rules also demonstrate how these clauses can be automatically generated by our tool; that is, instead of checking that each assignment and call is allowed as shown in the T-rules, the tool would add the variable or method names to the assignable and callable clauses of the generated specification.

The subsections that follow explain how the T-rules, in Figures 5.1 - 5.3, enforce the super-call invalidation and alias control rules from Chapters 2 and 3. Also, the definition of the T-rules demonstrate that our technique and the rules given in Chapters 2 and 3 can be checked and enforced statically. These typing rules are also necessary for the soundness of our technique for reasoning about and verifying the behavior of subclass methods without superclass code (i.e., the soundness of the programming logic defined in Section 4.6).

5.1.2 Formalizing the Alias Control Rules

In this subsection, we first review the definition of owner variable and describe how they are represented in the T-rules. We then explain how the T-rules formalize the Pivot Assignment, Owner Variable, and Actual Parameter Aliasing Rules. These three rules from Chapter 3 protect pivot objects and prevent unwanted aliasing and side-effects.

5.1.2.1 Owner variables

As explained in Chapter 3, the main principle in our alias control technique is that changes to the state of objects other than the receiver must be initiated through an owner variable visible in the current context. In our technique, non-pivot fields and local variables can only be temporary owners within a local context; a static analysis handles this by keeping track of temporary owner variables. The set O in the T-rules of Figures 5.1 - 5.3 is this set of temporary owner variables.

Recall (as defined in subsection 3.5.1) that a variable is an owner if it is the first (in time) to hold a reference to a newly created object. Therefore, a variable can only be an owner after it occurs on the left side of an assignment and the right side is a new object constructor call. This is formalized in the T-NewAssign rule of Figure 5.2, i.e., the variable reference (*vr*) from the left side of the assignment is added to the set of owner variables (O) because *vr* will be the first to hold a reference to the newly created object. Also, the T-NewAssign rule is the only rule that adds variable references to O. In contrast, the T-ExpAssign, T-CallAssign, and T-SupCallAssign rules of Figure 5.2 remove the

(T-Call)

$$\frac{\begin{array}{l} T = \mathit{whereMethodDecl}(\ \mathit{typeOf}(e0),\ m\ ),\\ \mathit{subst}(\mathit{rcvr}, T).m \in \mathit{calls}(U, n),\\ (\ e0 \in O \vee\ (\mathit{selfAssigns}(e0, T, m) \subseteq_a \mathit{assigns}(U, n))\ ),\\ (\ e \in O \vee\ (\mathit{parmAssigns}(e, T, m) \subseteq_a \mathit{assigns}(U, n))\ ),\\ \mathit{invariantOK}(e0, e, O, U, n),\ \mathit{aliasingOK}(e0, e, U, T, m), \end{array}}{(O, U, n) \vdash_t e0.m(e); \rightarrow_o O}$$

(T-SupCall)

$$\frac{\begin{array}{l} T = \mathit{whereMethodDecl}(\ \mathit{superOf}(U),\ m\ ),\ T{::}m \in \mathit{calls}(U, n),\\ \mathit{selfAssigns}(\texttt{this}, T, m) \subseteq_a \mathit{assigns}(U, n),\\ (\ e \in O \vee\ (\mathit{parmAssigns}(e, T, m) \subseteq_a \mathit{assigns}(U, n))\ ),\\ \mathit{invariantOK}(\texttt{this}, e, O, U, n),\ \mathit{aliasingOK}(\texttt{this}, e, U, T, m),\\ \mathit{okToSuperCall}(U, T, m) \end{array}}{(O, U, n) \vdash_t \texttt{super.}m(e); \rightarrow_o O}$$

(T-SupConstr)

$$\frac{\begin{array}{l} T = \mathit{superOf}(U),\ T{::}T \in \mathit{calls}(U, U)\ ),\\ \mathit{selfAssigns}(\texttt{this}, T, T) \subseteq_a \mathit{assigns}(U, U),\\ (\ e \in O \vee\ (\mathit{parmAssigns}(e, T, T) \subseteq_a \mathit{assigns}(U, U))\ ),\\ \mathit{invariantOK}(\texttt{this}, e, O, U, U),\ \mathit{aliasingOK}(\texttt{this}, e, U, T, T),\\ \mathit{noDownCalls}(U, T, T),\ !(e \equiv \texttt{this}) \end{array}}{(O, U, U) \vdash_t \texttt{super}(e); \rightarrow_o O}$$

Figure 5.1: T-rules for method calls.

variable reference on the left side of the assignment from O because such variables may not be the first to hold the object referenced[1]. Therefore, when non-pivot fields and local variables are owners, they are allowed to modify the object they reference, but only during the current execution of a method or until they are the left side of an assignment statement with a right side that is not a new object constructor call.

The T-If rule of Figure 5.3 is also interesting because it requires that a variable be an owner at the end of both branches of an if-statement in order to continue as a temporary owner. Similarly, the T-

1. In Java, this would be all assignments in which the right-hand side is not a new object constructor call.

$$\text{(T-CallAssign)}\quad \frac{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} e0\texttt{.}m\texttt{(}e\texttt{)}\texttt{;} \rightarrow_{\mathrm{o}} \mathrm{O},\quad (!\mathit{isField}(\mathit{vr}) \vee \mathit{vr} \in_{\mathrm{a}} \mathit{assigns}(\mathrm{U}, n)),\quad \mathit{okToAssign}(\mathit{vr})}{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} \mathit{vr}\texttt{=}e0\texttt{.}m\texttt{(}e\texttt{)}\texttt{;} \rightarrow_{\mathrm{o}} \mathrm{O} - \{\mathrm{vr}\}}$$

$$\text{(T-SupCallAssign)}\quad \frac{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} \texttt{super.}m\texttt{(}e\texttt{)}\texttt{;} \rightarrow_{\mathrm{o}} \mathrm{O},\quad (!\mathit{isField}(\mathit{vr}) \vee \mathit{vr} \in_{\mathrm{a}} \mathit{assigns}(\mathrm{U}, n)),\quad \mathit{okToAssign}(\mathit{vr})}{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} \mathit{vr}\texttt{=super.}m\texttt{(}e\texttt{)}\texttt{;} \rightarrow_{\mathrm{o}} \mathrm{O} - \{\mathit{vr}\}}$$

$$\text{(T-ExpAssign)}\quad \frac{(!\mathit{isField}(\mathit{vr}) \vee \mathit{vr} \in_{\mathrm{a}} \mathit{assigns}(\mathrm{U}, n)),\quad \mathit{okToAssign}(\mathit{vr})}{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} \mathit{vr}\texttt{=}e\texttt{;} \rightarrow_{\mathrm{o}} \mathrm{O} - \{\mathit{vr}\}} \quad \textbf{if } !(e \equiv \texttt{null})$$

$$\text{(T-NullAssign)}\quad \frac{(!\mathit{isField}(\mathit{vr}) \vee \mathit{vr} \in_{\mathrm{a}} \mathit{assigns}(\mathrm{U}, n))}{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} \mathit{vr}\texttt{=null;} \rightarrow_{\mathrm{o}} \mathrm{O} - \{\mathit{vr}\}} \quad \textbf{if } e \equiv \texttt{null}$$

$$\text{(T-NewAssign)}\quad \frac{\mathrm{T}\texttt{.}\mathrm{T} \in \mathit{callsOf}(\mathrm{U}, n),\quad (\ e \in \mathrm{O} \vee\ (\mathit{parmAssigns}(e, \mathrm{T}, \mathrm{T}) \subseteq_{\mathrm{a}} \mathit{assigns}(\mathrm{U}, n))\ ),\quad \mathit{invariantOK}(\texttt{this}, e, \mathrm{O}, \mathrm{U}, n),\quad (!\mathit{isField}(\mathit{vr}) \vee \mathit{vr} \in_{\mathrm{a}} \mathit{assigns}(\mathrm{U}, n)),}{(\mathrm{O}, \mathrm{U}, n) \vdash_{\mathrm{t}} \mathit{vr}\texttt{=new T(}e\texttt{)}\texttt{;} \rightarrow_{\mathrm{o}} \mathrm{O} \cup \{\mathit{vr}\}}$$

Figure 5.2: T-rules for assignment statements.

While rule requires that a variable be an owner whether or not the while-loop is executed. This handling of if- and while-statements is similar to the way Java checks that a field has been initialized prior to first use.

5.1.2.2 Formalizing the Pivot Assignment Rule

The five T-rules shown in Figure 5.2 formalize the Pivot Assignment Rule described in Chapter 3 (subsection 3.5.2). The purpose of this rule is to make sure that pivot fields and parameters always own the objects they reference. Pivot fields must always be owner variables because we want pivot objects of the receiver to be modifiable by all methods declared in a class. Similarly, when the object referenced by a formal parameter needs to be modified, that parameter must also be an owner variable

(T-Skip) $(O, U, n) \vdash_t \texttt{;} \rightarrow_o O$

(T-LocalDecl) $(O, U, n) \vdash_t \texttt{T x;} \rightarrow_o O$

(T-Return) $(O, U, n) \vdash_t \texttt{return } e\texttt{;} \rightarrow_o O$

(T-If)
$$\frac{(O, U, n) \vdash_t C1 \rightarrow_o O',\ (O, U, n) \vdash_t C2 \rightarrow_o O''}{(O, U, n) \vdash_t \texttt{if (}e\texttt{) } C1 \texttt{ else } C2 \rightarrow_o O' \cap O''}$$

(T-While)
$$\frac{(O, U, n) \vdash_t C \rightarrow_o O'}{(O, U, n) \vdash_t \texttt{while (}e\texttt{) } C \rightarrow_o O \cap O'}$$

(T-Seq)
$$\frac{(O, U, n) \vdash_t C1 \rightarrow_o O',\ (O', U, n) \vdash_t C2 \rightarrow_o O''}{(O, U, n) \vdash_t C1\ C2 \rightarrow_o O''}$$

Figure 5.3: T-rules that do not involve method calls or assignments.

because, otherwise, those side-effects would not be allowed by our technique. Therefore, pivot fields and parameters must be handled differently from other program variables, that is, our technique must make sure they can never reference objects they do not own. The predicate *okToAssign*(*vr*) (Figure 5.4) appears as an antecedent in the T-ExpAssign, T-CallAssign, and T-SupCallAssign rules of Figure 5.2; this predicate disallows the assignment when the target variable is a pivot field or parameter unless the target has a primitive type; primitive types cannot be aliased so such assignments can be allowed. Therefore, these rules do not allow assignments when the left side is a pivot field or parameter and the right side is an object reference[2].

However, an assignment is allowed when the target is a pivot field or parameter and the right side is either `null` (T-NullAssign rule) or a new object constructor call (T-NewAssign rule), i.e.,

2. The Pivot Assignment Rule of Chapter 3 allows references with an immutable type to be assigned to parameters and pivot fields (subsection 3.5.2); however, for simplicity, we do not consider immutable types in our formalization in this chapter.

---

invariantOK(*rcvr*, *arg*, O, U, *n*) =
  isOwner(*rcvr*, O, U) ∧ (*typeOf*(*arg*) ∉ *TypeId* ∨ *isOwner*(*arg*, O, U))

aliasingOK(*rcvr*, *arg*, U, T, *m*) = *assigns*(T, *m*) = { }
  ∨ (!(*rcvr* ≡ *arg*) ∧ (!*isPivot*(*arg*, U) ∨ !(*rcvr* ≡ `this`))
    ∧ (!*isPivot*(*rcvr*, U) ∨ !(*arg* ≡ `this`)))

okToSuperCall(S, T, *m*) =
  validInvariant(S, T, *m*) ∧ *validCalls*(S, T, *m*)

noDownCalls(S, T, *m*) =
  (∀ *n* ∈ *setOfMethodsIn*(T) :
    `this.`*n* ∉ *calls*(T, *m*) ∨ *noOverriddenMethods*(S, T))
  ∧ (∀ U`.`*n* ∈ *calls*(T, *m*) : *noParmDowncalls*(S, T, U, *n*))

isPivot(*e*, T) = (*isField*(*e*) ∧ *e* ∈ *pivotFieldsIn*(T))

okToAssign(*vr*, T) =
  typeOf(*vr*) ∉ *TypeId* ∨ (!*isPivot*(*vr*, T) ∧ !(*vr* ≡ `p`))

selfAssigns(*rcvr*, T, *m*) = { `this.`*g* | `this.`*g* ∈ *assigns*(T, *m*) }[`this`←*rcvr*]
  ∪ { `this.`*g*`.`*x* | `this.`*g*`.`*x* ∈ *assigns*(T, *m*) }[`this`←*rcvr*]

parmAssigns(*arg*, T, *m*) = { `p.`*g* | `p.`*g* ∈ *assigns*(T, *m*) }[`p`←*arg*]

subst(*rcvr*, T) = **if** *rcvr* ≡ `this` **then** `this` **else** T

Figure 5.4: Top level functions used in the T-Rules; the related helper functions are in Figures 5.5 and 5.6.

---

okToAssign(*vr*) is not a premise in either of these rules. Since no other assignments to pivots or parameters are allowed, these variables will always own the object they reference.

The Pivot Assignment Rule, through the rules of Figure 5.2, also ensure that each pivot object is referenced by no more than one pivot field (since only one field can be the first to contain a specific object reference). However, it is safe to allow pivot objects to be aliased by non-owner variables since changes to pivot objects cannot be initiated through a non-owner variable. Thus, as required for soundness, modifications of any particular pivot object are only allowed through a single pivot field.

$$isOwner(vr, O, U) = (vr \in O \ \vee \ vr \equiv \texttt{this} \ \vee \ vr \equiv \texttt{p} \ \vee \ isPivot(vr, U))$$

$$validInvariant(S, T, m) =$$
$$(\forall \ \texttt{this}.f \in_a assigns(T, m) : \texttt{this}.f \notin accessed(invOf(\texttt{TEnv}(S))))$$
$$\wedge \ (\forall \ \texttt{this}.f.g \in_a assigns(T, m) : \texttt{this}.f.g \notin accessed(invOf(\texttt{TEnv}(S))))$$

$$validCalls(S, T, m) = validSelfCalls(S, T, m) \ \wedge \ validObjectCalls(S, T, m)$$
$$\wedge \ (\forall \ U::n \in calls(T, m) : validCalls(S, U, n))$$

$$validSelfCalls(S, T, m) =$$
$$(\forall \ \texttt{this}.n \in calls(T, m) : \ noAddSideEffects(S, T, n))$$

$$validObjectCalls(S, T, m) =$$
$$(\forall \ U.n \in calls(T, m) : \ noParmAddSideEffects(S, T, m, U, n))$$

$$noAddSideEffects(S, T, m) =$$
$$(\forall f \in setOfFieldsIn(S), g \in VarId :$$
$$\texttt{this}.f \notin_a assigns(T, m) \ \wedge \ \texttt{this}.f.g \notin_a assigns(T, m))$$

$$noParmDowncalls(S, T, U, n) =$$
$$getParmType(U, n) \notin TypeId \ \vee \ !(T \leq getParmType(U, n))$$
$$\vee \ noOverriddenMethods(S, T)$$

$$noParmAddSideEffects(S, T, m, U, n) =$$
$$getParmType(U, n) \notin TypeId \ \vee \ !(T \leq getParmType(U, n))$$
$$\vee \ !(parmAssigns(\texttt{this}, U, n) \subseteq_a assigns(T, m))$$
$$\vee \ (\forall g \in setOfFieldsIn(S) : \texttt{p}.g \notin_a assigns(T, m))$$

$$noOverriddenMethods(S, T) =$$
$$(\forall m \in setOfMethodsIn(T) : \ ! \ isOverridden(S, m))$$

Figure 5.5: Helper functions used indirectly in the T-Rules by functions in Figure 5.4.

5.1.2.3 Formalizing the Owner Variable Rule

Another important part of our technique for controlling side-effects and aliasing is our restrictions on the left side of assignment statements. As explained in subsections 1.6.6, 2.4.3, and 3.5.7, direct

$$isField(\ e\ ) = (\exists f \in VarId\ :\ e \equiv \texttt{this}.f\ )$$

$$vr \in_a A\ =\ (\ vr \in A \quad \vee\ (\exists\ pre.g \in A\ :\ vr \in_a datagroupOf(pre, typeOf(pre), g))\ )$$

$$datagroupOf(pre, T, g) = \{\ pre.f \mid f \in allFieldsIn(T)\ \wedge\ g \in inOf(lookupField(T, f))\ \} \cup \{\ pre.f.x \mid f \in allFieldsIn(T)\ \wedge\ (f.x, g) \in mapsOf(lookupField(T, f))\ \}$$

$$A \subseteq_a B = (\forall\ vr \in_a A\ :\ vr \in_a B\ )$$

$$allFieldsIn(T) = \{ f \mid T \leq U\ \wedge f \in setOfFieldsIn(U)\ \}$$

$$setOfFieldsIn(T) = \{ f \mid fieldsOf(\texttt{TEnv}(T))\ (f) \neq \texttt{undef}\ \}$$

$$pivotFieldsIn(T) = \{\ \texttt{this}.f \mid f \in allFieldsIn(T) \wedge\ mapsOf(\ fieldsOf(\texttt{TEnv}(U))\ (f)\ ) \neq \{\ \}\ \}$$

$$allMethodsIn(T) = \{\ U.m \mid m \in MethId\ \wedge\ lookupMethod(T, m) \neq \texttt{undef} \wedge\ U = whereMethodDecl(T, m)\ \}$$

$$setOfMethodsIn(T) = \{\ m \mid methodsOf(\texttt{TEnv}(T))\ (m) \neq \texttt{undef}\ \}$$

Figure 5.6: Additional helper functions used in the T-Rules.

assignment to fields of objects other than the receiver is not allowed. That is, our technique does not allow the state of an object, other than the receiver, to be changed except through an object-call; this is enforced by the restricted syntax of the assignment statement in Java-C[3].

However, based on the Owner Variable Rule from Chapter 3 (subsection 3.5.1), our technique must also restrict object-calls. That is, an object-call that assigns to fields of its receiver must not be allowed unless the receiver expression is an owner variable. For example, if `x.m()` modifies the state of object `x`, then `x` must be an owner variable. Similarly, if a method modifies the state of the object referenced by a formal parameter, then the corresponding actual parameter must be an owner[4]. For

3. JML does not have this syntactic restriction so violations would be flagged by the JML type checker.

example, if `this.m(y)` modifies the state of object `y`, then `y` must be an owner variable. These restrictions on method calls enforce the Owner Variable Rule given in Chapter 3.

The Owner Variable Rule of subsection 3.5.1 is formalized by the T-Call, T-SupCall, and T-SupConstr rules of Figure 5.1 and the T-NewAssign rule of Figure 5.2. All of these rules make sure that the actual parameter is an owner variable whenever the method modifies the corresponding argument object; this is formalized through the predicate *invariantOK* (Figure 5.4) that appears in the antecedents of these rules. This predicate requires that each actual parameter (e.g., *e0* and *e*) be an owner variable (*isOwner*) unless it is not a reference type.

Furthermore, assignments to fields of a formal parameter object `p` must be done through an object-call, and `p` must be the receiver in that call. Therefore, a formal parameter `p` must be an owner whenever the method specification allows the object referenced by `p` to be modified; this is necessary because, otherwise, the specified side-effects would not be allowed by our technique. Furthermore, in our technique, this ownership property of variables is not transferable except to a parameter in a method call. For example, if `x` is an owner variable, then assigning `x` to `v` will not make `v` an owner variable; in fact `v` would not be an owner based on the T-ExpAssign rule (Figure 5.2).

In our technique, ownership is transferred temporarily from the actual parameters to the formal parameters during a method call. As described above, when a formal parameter needs to be an owner, the corresponding actual parameter must also be an owner variable. Therefore, the formal parameter becomes an owner during execution of a method and, as required for soundness, the changes made to the corresponding argument object are initiated indirectly through an owner variable. Ensuring that the actual parameters are owners when necessary is formalized in these rules through the predicate *invariantOK* of Figure 5.4 and indirectly through predicate *isOwner* of Figure 5.4.

Note, however, that the T-rules, with predicate *invariantOK* as an antecedent, are more restrictive than the Owner Variable Rule given in Chapter 3. That is, these T-rules require that all actual parameters be owner variables when they are a reference type, whereas in Chapter 3, this was only required when the called method had permission to change the state of the corresponding argument object. The reason for the more restrictive T-rules is because the only objects that are guaranteed not to have a invalidated type invariant are those referenced by owner variables. On the other hand, if the verifier can prove that the type invariant holds for an object referenced by a non-owner variable (variables containing a read-only reference), then it is safe to pass that object as a parameter as long as the called method does not change the state of that argument object.

---

4. In Java, a new object constructor call would also be a valid actual parameter; this case is not included in our formalization here even though it is part of the Owner Variable Rule of Chapter 3 because the Java-C syntax of expressions and assignment statements (Figure 4.5) requires that all objects be referenced by a variable, i.e., new object constructor calls can only occur in an assignment statement.

5.1.2.4 Formalizing the Actual Parameter Aliasing Rule

Only assignments to fields of the receiver, pivot objects, and objects referenced by a formal parameter have to be specified in the assignable clause since the other owners will be temporary owners that reference a newly created object (see subsections 5.1.2.1 and 5.1.7). However, in general, a formal parameter can be an alias of a pivot field (or the receiver `this`) during a self-call; this is a potential concern when the state of the aliased object is changed through one of these owner variables and accessed through the other. Therefore, when there are side-effects, our technique, through the Actual Parameter Aliasing Rule of subsection 3.5.3, does not allow more than one variable, visible in the same context, to own the same object during a method call. This rule is formalized through the *aliasingOk* predicate that appears as an antecedent in the T-Call, T-SupCall, and T-SupConstr rules of Figure 5.1. The *aliasingOk* predicate prevents a formal parameter from being an alias of the receiver or of a pivot field of the receiver when there are side-effects.

Also, an internal object can only be a non-pivot if none of the methods of the enclosing object access the state of that object. Thus aliasing of non-pivot objects does not cause problems since the state is not accessed. Furthermore, two pivot fields cannot be aliases of the same object. Thus assignments can be specified and checked precisely and modularly since the only side-effects allowed by our technique (other than to newly created objects) are assignments to fields of the receiver, fields of a pivot object, or fields of a formal parameter; also, these are the only assignments that have to be specified in the assignable clause.

5.1.3 Formalizing the Invalidation Rules

As mentioned previously, our technique has to invalidate super-calls that have unverifiable side-effects or may no longer satisfy their specification. This is done through the predicate *okToSuperCall* of Figure 5.4; this predicate is an antecedent in the T-SupCall rule of Figure 5.1 to ensure that the Additional Side-Effects and Invariant Invalidation Rules are not violated. If the superclass method has not been invalidated, then the call is allowed.

In the predicate *okToSuperCall*, parameter S denotes the subtype and T denotes the supertype. The predicate *okToSuperCall* makes sure that a super-call is to a method that has not been invalidated by the new subclass. The predicate *validInvariant* of Figure 5.5 specifies that a method can only be super-called if the superclass method has not been invalidated by the Invariant Invalidation Rule of Chapter 2. Similarly, the predicate *validCalls* of Figure 5.5 allows the super-call if the method has not been invalidated by the Additional Side-Effects Invalidation Rule of Chapter 2.

Specifically, predicate *validInvariant* of Figure 5.5 says that a super-call is invalid if it is allowed to modify a field that is constrained by a subclass invariant. This predicate formalizes the Invariant Invalidation Rule by making sure the super-call does not invalidate any of the subclass parts of the type invariant (as explained in subsection 2.4.1).

Predicate *validCalls* of Figure 5.5 says that a super-call is invalid in the context of a new subclass if it makes a downcall to a method that assigns to subclass fields. This predicate formalizes the Additional Side-Effects Invalidation Rule by making sure the super-call does not have unverifiable side-effects (as explained in subsection 2.2.3).

The T-SupConstr rule is similar except that a superclass constructor is not allowed to make downcalls, i.e., the predicate *noDownCalls* appears in the antecedent of this rule. Disallowing all downcalls is more conservative than the rule given in Chapter 2, i.e., the Constructor Initialization Invalidation Rule (subsection 2.4.5) only invalidates the superclass constructor if it makes a downcall that accesses a subclass field (or a superclass field that is not accessed by the overridden method). However, the Constructor Initialization Invalidation Rule would need to use the accessible clause, and we are leaving the semantics and use of the accessible clause as future work (see subsection 4.1.2.2).

In summary, the *okToSuperCall* and *noDownCalls* predicates are used in the T-SupCall and T-SupConstr rules to ensure that the invalidation rules from Chapter 2 are not violated. Predicate *okToSuperCall* uses *validInvariant* and *validCalls* to ensure that the Invariant and Additional Side-Effects Invalidation Rules are not violated. Predicate *noDownCalls* ensures that these invalidation rules and the Constructor Initialization Invalidation Rule are not violated by superclass constructor calls.

5.1.3.1 Handling aliasing of the receiver in this-argument calls

The predicate *validCalls* (Figure 5.5) has to identify all downcalls with additional side-effects in order to prevent them. The second conjunct of *validCalls* invokes predicate *validObjectCalls* and is necessary because our technique allows dynamic aliasing through formal parameters. For example, in a this-argument call, a formal parameter of the called method could be an alias of the current receiver. Therefore, an object-call on that formal parameter in the called method would be a call on the current receiver (and would have the same effect as a self-call, i.e., could be a downcall). However, since we are only interested in downcalls that assign to subclass fields, it is only necessary that side-effects to fields of that formal parameter be checked; this is formalized in the predicate *noParmAddSideEffects*. Predicate *noParmAddSideEffects* does not allow the super-call if a subclass field could be changed by an object-call with a formal parameter that could be an alias of the current receiver.

The receiver is nonspecific when we list object-calls in the callable clause, i.e., the actual parameters are not specified to keep the specification more abstract; thus we have to assume that the formal parameter is an alias of the receiver whenever that is a possibility. However, the formal parameter will not always be an alias of the current receiver; nonetheless, this check is necessary in case the object-call is a this-argument call and so the check can be done modularly (i.e., without checking the whole program). However, if the verifier can prove that the receiver is not aliased in the context of the called method, then these possible downcalls can be disregarded[5]. Nonetheless, without superclass code, we have to assume that the receiver could be aliased and that there could be downcalls

that assign to subclass fields when that is a possibility. For the same reason, a similar conjunct to check object-calls is also needed in predicate *noDownCalls* of Figure 5.4, i.e., the second conjunct.

It is unclear, in practice, how often object-calls will be mistakenly considered this-argument calls by our checking rules. However, *noParmAddSideEffects* of Figure 5.5 also checks, in the third disjunct, whether the assignments to fields of the formal parameter would be permitted if the object-call were a this-argument call. That is, the object-call is only considered if the side-effects to the formal parameter are allowed when `this` is the corresponding actual parameter (as would be checked in the T-Call rule of Figure 5.1, i.e., the last disjunct of function *noParmAddSideEffects*).

5.1.4 Formalizing the Predicate Clause Access Rule

If a field `this.`*f* is the receiver in an object-call, then the specification for the method making that object-call may have to access the state of object `this.`*f*. However, since concrete fields are not in scope in public specifications, the method's behavior would have to be specified in predicate clauses (e.g., requires or ensures clauses) that indirectly access, through a model field, the state of object `this.`*f*. The Predicate Clause Access Rule says that such objects have to be pivots because the method's behavior depends on the state of that object.

We assume, in our formalization, that the state of an object is accessed anytime `this.`*f* is an actual parameter in a method call, i.e., `this.`*f* must be a pivot field if it is an actual parameter in a method call. Therefore, the Predicate Clause Access Rule is formalized by the *invariantOk* predicate (see also subsection 5.1.8.2) by requiring that all actual parameters be owner variables; thus, in this case, fields that are actual parameters must be pivot fields.

5.1.5 Callback Cycles

Our type system will not consider or formalize the rules involving callback cycles, i.e., the Callback Cycle Overriding and Callback Cycle Invalidation Rules of Section 2.5; this is because our verification logic is a partial correctness rather than the total correctness logic. The purpose of these two rules is to make sure that the code for all methods involved in a correctness proof are available, i.e., all methods involved in a callback cycle have to be available and thus no callback cycle can involve both superclass and subclass methods.

In addition, checking these two callback cycle rules requires that all methods directly or indirectly called be listed in the callable clause, i.e., the transitive closure of the calling relation between methods would have to be computed and considered in a checking rule. In contrast, the rest of our rules only need to know the methods directly called and only those that could result in a call on the current receiver. Furthermore, unlike the rest of the rules, the checking of the callback cycle rules cannot be

---

5. Since object-calls on newly created objects cannot be self-calls, they can be ignored by our formal system. Therefore, the semantics of the callable clause only requires that calls on objects existing in the pre-state be listed or checked (see subsection 5.1.7).

done using only the protected code contract because of visibility restrictions. We leave the checking of these rules and a total correctness verification logic as future work.

5.1.6 Unoverrideable Methods

For simplicity, we also leave the handling of unoverrideable methods as future work. In particular, the Java-C syntax (Figure 4.5) and our assumptions (subsection 1.6.6) do not allow static, private, or final methods. However, we believe that our technique can be extended to include these unoverrideable methods.

For example, our technique is primarily concerned with self-calls and super-calls that (may) make downcalls. Since static and private methods are unoverrideable, calls to these methods do not have to be specified in the callable clause as long as the calls made indirectly through these methods are listed in the callable clause (as described in subsection 2.6.4).

We are also concerned with object-calls on fields because such fields have to be pivots based on the Predicate Clause Access Rule (subsection 5.1.4). However, our technique does not allow unoverrideable methods with side-effects to be invoked in object-calls (see subsection 2.4.3) and static methods do not have a receiver; thus calls of static methods, private methods, and final methods with side-effects would not have to be considered when checking the Predicate Clause Access Rule. Only object-calls on final methods without side-effects would be allowed or have to be considered (i.e., specified in the callable clause).

In summary, our technique would not require that static or private methods be listed in the callable clause, as long as the non-private instance methods indirectly called are listed. Also, final methods without side-effects would have to be listed in the callable clause for use when checking the Predicate Clause Access Rule.

5.1.7 Checking Assignable and Callable Clauses

The T-rules given in Figures 5.1 - 5.3 also specify how the method body is checked to make sure it satisfies its assignable and callable clauses. Figure 5.1 defines the T-rules for checking method calls, Figure 5.2 contains the T-rules for checking assignment statements, and Figure 5.3 defines the T-rules that do not involve assignments or method calls.

In all of the T-rules, the U denotes the static type of the receiver and *n* denotes the method name where the statement being checked occurs. The functions *assigns* and *calls* of Figure 4.11 are used to extract the sets of assignable fields and callable methods from the assignable and callable clauses. For example, when checking method *n* declared in type U, the set of assignable fields is retrieved using the function *assigns*(U, *n*); similarly, the set of callable methods is retrieved using function *calls*(U, *n*). Also, when checking method call statements, the *assigns* function is used in the T-rules for extracting the assignable field sets from the specification of the called method; the T-Call rule of Figure 5.1 is an example.

The checking specified in the T-rules does not consider the preconditions of specification cases; thus, to be sound, these rules only allow assignments that are common to all specification cases, e.g., the *assigns* function of Figure 4.11 takes the intersection of the assignable sets from each method declaration in the inheritance hierarchy.

The main idea of our checking rules is that the fields assigned and methods called must be permitted by the assignable and callable clauses of the method specification, i.e., they must be allowed by sets *assigns*(U, *n*) and *calls*(U, *n*). The checking of the assignable clauses is done using the auxiliary functions *selfAssigns* and *parmAssigns* of Figure 5.4. The functions *selfAssigns* and *parmAssigns* of Figure 5.4 create specific subsets of the set of all assignable fields; that is, *selfAssigns* creates the subset containing the assignable fields of the receiver and *parmAssigns* creates the subset of assignable fields for the formal parameter `p`; these functions also substitute the actual parameter for the formal parameter in these sets of fields, i.e., *rcvr* is substituted for `this` and *arg* for `p`. The T-rules of Figures 5.1 and 5.2 use *selfAssigns* and *parmAssigns* when checking whether the calling method is allowed to make the assignments made by the called method or constructor.

The assignable clause specifies the assignments allowed to variables that existed in the pre-state. That is, assignments to variables that did not exist in the pre-state are allowed and do not have to be checked. For example, the semantics of the assignable clause does not restrict assignments to local variables because local variables did not exist in the pre-state. Therefore, assignments to fields of an object that did not exist in the pre-state are also allowed and do not have to be included in the specification; this is formalized by the conditions $e0 \in O$ and $e \in O$ in the T-Call, T-SupCall, T-SupConstr, and T-New rules of Figures 5.1 and 5.2. For example, in the T-Call rule, if the receiver is a newly created object, then all assignments to fields of that object are not checked (recall that O is the set of temporary owner variables that reference newly created objects). Similarly, assignments to fields of a formal parameter are not restricted if the corresponding actual parameter is a newly created object; this is formalized by the condition $e \in O$ in the antecedents of these rules. Therefore, assignments to fields of new objects are not included in the assignable field sets; hence, these subsets do not have to be checked when the corresponding actual parameter is a newly created object, i.e., when *e* is a member of O.

Figure 5.2 defines the T-rules for checking assignment statements. These rules must make sure the target variable is assignable when it is a field of the receiver (e.g., `this.`*f*). As described above, assignments to local variables or parameters can be ignored since they did not exist in the pre-state; this is formalized in the disjunct involving expressions *isField*(*vr*) and $vr \in_a assigns(U, n)$, i.e., if the target is a field, then that field must be assignable. Notice that this disjunct appears as an antecedent in each of the rules of Figure 5.2. Also, the self-assignments are not included in the rule for new object constructor calls (T-NewAssign) because these assignments would be to fields of a newly created object.

The T-Call rule of Figure 5.1 determines whether or not a call is allowed; this rule uses the function *subst* of Figure 5.4 in the textual substitution of the receiver parameter when checking whether the call is a member of the callable methods. In particular, *subst* substitutes the actual parameter when it is `this`, otherwise it substitutes the type of the actual parameter. Thus when the actual parameter is a field or local variable, the receiver is replaced by the type; as described above, this indicates an object-call on some variable other than the receiver. Notice that we require all direct method calls be listed in the callable clause but we do not check indirect calls.

5.1.8 Checking the Consistency of Class Specifications

In this subsection, we formalize a few of the rules that do not involve the checking of method statements. For example, we formalize the rules used for checking that the class definition and specification are well-formed. The functions defined in Figure 5.7 are applied to each class declaration in the hierarchy.

5.1.8.1 Formalizing the overriding rules

The predicate *requiredOverrides* of Figure 5.7 requires that a method declared in the superclass be overridden if it is allowed to modify a subclass field, i.e., if it could have additional side-effects. This predicate formalizes the Additional Side-Effects Overriding Rule (subsection 2.2.2).

Predicate *requiredOverrides* is also used to formalize the Invariant Overriding Rule (subsection 2.4.2), i.e., it requires an override to prevent unoverridden superclass methods from being invoked when they could invalidate a subclass invariant. Note that this predicate is necessary because a self-call to an unoverridden superclass method can have the same kinds of downcall problems as super-calls; thus our technique has to use overrides to prevent these methods from being invoked.

5.1.8.2 Formalizing the Pivot Declaration Rule

The Pivot Declaration Rule requires that an object be a pivot if its state is accessed by the right hand side of a represents clause. This rule is satisfied when the dependency relationships between concrete fields and model fields are properly specified in data group clauses. Predicate *validRepresents* requires that every model field of a type have an admissible representation, i.e., have data group clauses that reflect the relationships specified in the represents clause. An admissible representation is one in which every field accessed by the right side of a model field's represents clause is a member of that model field's data group; predicate *admissibleRep* enforces this requirement. The function *accessed* was defined in Chapter 3, Figure 3.6.

5.1.8.3 Formalizing the Assignable Data Group Rule

The predicate *validAssignable* checks that all model fields are listed in an assignable clause whenever any of the concrete members of its data group are assignable. This is necessary because data groups may overlap, i.e., two model fields may depend on the same concrete field. Therefore, the specifier could, in error, list only one of the model fields even though both model fields could change.

$$
\begin{array}{l}
\mathit{requiredOverrides}(\mathrm{T}) = \\
\quad (\forall\ \mathrm{U}.n \in \mathit{allMethodsIn}(\mathit{superOf}(\mathrm{T}))\ :\\
\quad\quad \mathit{isOverridden}(\mathrm{T}, n)\\
\quad\quad \vee\ \ (\ \mathit{noAddSideEffects}(\mathrm{T}, \mathrm{U}, n)\ \wedge\ \mathit{validInvariant}(\mathrm{T}, \mathrm{U}, n)\ )\ )
\end{array}
$$

$$
\begin{array}{l}
\mathit{validRepresents}(\mathrm{T}) = \\
\quad (\forall\ g \in \mathit{setOfFieldsIn}(\mathrm{T})\ :\\
\quad\quad !\mathit{isModelField}(\mathit{lookupField}(\mathrm{T}, g))\ \vee\ \mathit{admissibleRep}(\mathrm{T}, g)\ )
\end{array}
$$

$$
\begin{array}{l}
\mathit{validAssignable}(\mathrm{T}) = \\
\quad (\forall\ m \in \mathit{setOfMethodsIn}(\mathrm{T}), g \in \mathit{allFieldsIn}(\mathrm{T})\ :\\
\quad\quad (\forall\ vr \in_{\mathrm{a}} \mathit{assigns}(\mathrm{T}, m)\ :\\
\quad\quad\quad vr \notin_{\mathrm{a}} \mathit{datagroupOf}(\mathrm{T}, g) \vee \texttt{this}.g \in_{\mathrm{a}} \mathit{assigns}(\mathrm{T}, m)\ )\ )
\end{array}
$$

$$
\begin{array}{l}
\mathit{validMethodSpecs}(\mathrm{T}) = \\
\quad (\forall\ m \in \mathit{setOfMethodsIn}(\mathrm{T})\ :\\
\quad\quad (\forall\ \ \mathit{pre}.\mathit{id} \in \mathit{assigns}(\mathrm{T}, m)\ :\ \ \mathit{validStoreRef}(\ \mathit{pre}.\mathit{id}\ )\ ))
\end{array}
$$

$$
\begin{array}{l}
\mathit{validStoreRef}(\ \mathit{pre}.\mathit{id}\ ) = \\
\quad (\ \mathit{pre} \equiv \texttt{this}\ \vee\ \mathit{pre} \equiv \texttt{p}\ )\ \wedge\ \mathit{isModelField}(\mathit{lookupField}(\mathit{typeOf}(\mathit{pre}), \mathit{id})
\end{array}
$$

$$
\mathit{isOverridden}(\mathrm{T}, m) = \ \mathit{methodsOf}(\texttt{TEnv}(\mathrm{T}))\ (m) \neq \texttt{undef}
$$

$$
\begin{array}{l}
\mathit{admissibleRep}(\mathrm{T}, g) = \\
\quad\ (\forall\ \texttt{this}.f \in \mathit{accessed}(\mathit{repOf}(\mathit{lookupField}(\mathrm{T}, g))\ :\\
\quad\quad\ g \in \mathit{inOf}(\mathit{lookupField}(\mathrm{T}, f))\ )\\
\quad \wedge\ (\forall\ \texttt{this}.f.x \in \mathit{accessed}(\mathit{repOf}(\mathit{lookupField}(\mathrm{T}, g))\ :\\
\quad\quad\ (f.x, g) \in \mathit{mapsOf}(\mathit{lookupField}(\mathrm{T}, f))\ )
\end{array}
$$

Figure 5.7: Functions used to check that the specifications are well-formed and consistent. Function *accessed* was defined in Figure 3.6.

This predicate ensures that whenever a concrete field is assignable, all model fields that depend on that concrete field are listed in an assignable clause, i.e., predicate *validAssignable* ensures that the Assignable Data Group Rule (subsection 3.4.3) is satisfied.

5.1.8.4 Formalizing the Assignable Clause Rule

Note also that the JML-C syntax (Figure 4.6) only allows store references if they have a specific form, e.g., `this.g` or `p.g`. Furthermore, in a public assignable clause, the prefix variable has to be a parameter (i.e., `this` or `p`) since fields and other local variables are not in scope in a public specification. Also, the suffix has to be a model field since the concrete fields of an object are not in scope in the public specification. The predicate *validStoreRef* formalizes this property of public assignable clauses and ensures that the Assignable Clause Rule (subsection 3.3.5) is not violated, i.e., fields of model fields cannot be listed in the assignable clause.

5.2 Soundness

A deductive system is sound if its axioms are always true and every derivable assertion is true. Thus, to be of any practical value, a deductive system must be sound and the soundness must be determined in relation to the intended semantics. For example, a deductive system for proving properties of programs must be sound with respect to the semantics of the programming language. This section presents a proof of the soundness of our programming logic, given in Figures 4.20 through 4.23, with respect to the operational semantics of Figures 4.16 through 4.19. The soundness of our programming logic also proves that our technique for controlling aliasing and verifying subclass methods without superclass code is also sound since our verification logic depends on the rules given in Chapters 2 and 3 and it uses only the superclass and subclass specifications to verify the correctness of subclass code.

We begin in subsection 5.2.1 by proving some properties needed for behavioral subtyping, i.e., relationships between superclass and subclass specifications. Subsection 5.2.2 proves some properties of our model of the program state and the relationship between variable assignment and expression substitution. In subsection 5.2.3, we prove the Valid Invariant Theorem; this theorem proves that, in our technique, a superclass method can establish the run-time type invariant of the receiver by establishing the invariant of the static type of the receiver. Subsection 5.2.4 proves the soundness of our alias control technique, i.e., that no pair of owner variables can reference (be aliases of) the same object. In subsection 5.2.5, we prove that super-calls, in our technique, do not have additional side-effects, i.e., do not modify subclass fields. Subsection 5.2.6 proves that our axioms are valid and that our inference rules preserve validity.

5.2.1 Relationships Between Superclass and Subclass Specifications

In this subsection, we show that the relationships between superclass and subclass specifications have been formalized so that subclasses have to be behavioral subtypes of their superclasses. In particular, subclasses inherit and therefore must satisfy the combined superclass and subclass specifications. These lemmas will be used in the proof that our verification logic is sound, i.e., they will be used in various parts of the proofs given in the subsections that follow.

Lemma 5.1 (Invariant Clause): Let T1, T $\in$ *TypeId*.

If T1 ≤ T, **then** *inv*(T1) ⇒ *inv*(T)

Proof:

The proof is by induction on the number of superclasses of T1 in the class hierarchy between T1 and T.

Basis: $k = 0$, i.e., T1 = T.

inv(T1)

⇒ < T1 = T >

inv(T)

Induction Step: Let k = N and T1, T2, ..., Tk, T ∈ *TypeId*.

The induction hypothesis asserts that

If $k < N \wedge$ T1 < T2 ... < Tk < T,

then *inv*(T1) ⇒ *inv*(T).

inv(T1)

⇒ < by the induction hypothesis, *inv*(T1) ⇒ *inv*(Tk) >

inv(Tk)

⇒ < definition of *inv* of Figure 4.11 >

invOf(Tk) && *inv*(*superOf*(Tk))

⇒ < *superOf*(Tk) = T from the definition of *superOf* in Figure 4.9 >

invOf(T1) && *inv*(T)

⇒ < logic >

inv(T)

■

Lemma 5.2 (Requires Clause): Let T1, T ∈ *TypeId*.

If T1 ≤ T ∧ *methodsOf*(`TEnv`(T1)) (*m*) ≠ `undef` ∧ *methodsOf*(`TEnv`(T)) (*m*) ≠ `undef`,

then *req*(T, *m*) ⇒ *req*(T1, *m*)

Proof:

The proof is by induction on the number of superclasses of T1 in the class hierarchy between T1 and T that also contain a declaration of *m*.

Basis: $k = 0$, i.e., T = T1.

req(T, *m*)

⇒ < T1 = T >

req(T1, *m*)

Induction Step: Let k = N and T1, T2, ..., Tk, T ∈ *TypeId*.

The induction hypothesis asserts that

If $k < N \wedge$ T1 < T2 ... < Tk < T

∧ *methodsOf*(`TEnv`(*tid*)) (*m*) ≠ `undef` for *tid*=T1, T2, ..., Tk, T,

then *req*(T, *m*) ⇒ *req*(T1).

req(T1, *m*)

$\Leftarrow$ < by the induction hypothesis >

req(Tk, *m*)

$\Leftrightarrow$ < definition of *req* of Figure 4.11 >

reqOf(*lookupMethod*(Tk, *m*)) || *req*(*superOf*(Tk), *m*)

$\Leftrightarrow$ < *superOf*(... *superOf*(Tk)...) = T from the definition of *superOf* in Figure 4.9 and *req* in Figure 4.11 and from the premise of the lemma, i.e., that *m* is declared in T >

reqOf(Tk, *m*) || *req*(T, *m*)

$\Leftarrow$ < logic >

req(T, *m*)

■

Lemma 5.3 (Ensures Clause): Let T1, T $\in$ *TypeId*.

If T1 $\leq$ T $\wedge$ *methodsOf*(`TEnv`(T1)) (*m*) $\neq$ `undef` $\wedge$ *methodsOf*(`TEnv`(T)) (*m*) $\neq$ `undef`,

then *ens*(T1, *m*) $\Rightarrow$ *ens*(T, *m*)

Proof:

The proof is by induction on the number of superclasses of T1 in the class hierarchy between T1 and T that also contain a declaration of *m*.

Basis: *k* = 0, i.e., T1 = T.

ens(T1, *m*)

$\Rightarrow$ < T1 = T >

ens(T, *m*)

Induction Step: Let k = N and T1, T2, ..., Tk, T $\in$ *TypeId*.

The induction hypothesis asserts that

If *k* < N $\wedge$ T1 < T2 ... < Tk < T

$\wedge$ *methodsOf*(`TEnv`(*tid*)) (*m*) $\neq$ `undef` for *tid*=T1, T2, ..., Tk, T,

then *ens*(T1, *m*) $\Rightarrow$ *ens*(T, *m*).

ens(T1, *m*)

$\Rightarrow$ < by the induction hypothesis, *ens*(T1, *m*) $\Rightarrow$ *ens*(Tk, *m*) >

ens(Tk, *m*)

$\Rightarrow$ < definition of *ens* of Figure 4.11 >

postCond(*lookupMethod*(Tk, *m*)) && *ens*(*superOf*(Tk), *m*)

$\Rightarrow$ < logic >

ens(*superOf*(Tk), *m*)

$\Rightarrow$ < *superOf*(... *superOf*(Tk)...) = T from the definition of *superOf* in Figure 4.9 and *ens* in Figure 4.11 and from the premise of this lemma, i.e., that *m* is declared in T >

ens(T, *m*)

■

Lemma 5.4 (Assignable Clause): Let T1, T ∈ *TypeId*.

If T1 ≤ T ∧ *methodsOf*(`TEnv`(T1)) (*m*) ≠ `undef` ∧ *methodsOf*(`TEnv`(T)) (*m*) ≠ `undef`,

then *assigns*(T1, *m*) ⊆ *assigns*(T, *m*)

Proof:

The proof is by induction on the number of superclasses of T1 in the class hierarchy between T1 and T that also contain a declaration of *m*.

Basis: *k* = 0, i.e., T1 = T.

assigns(T, *m*) ⊆ *assigns*(T, *m*)

⇒ < T1 = T >

assigns(T1, *m*) ⊆ *assigns*(T, *m*)

Induction Step: Let k = N.

The induction hypothesis asserts that

if *k* < N ∧ T1 < T2 ... < Tk < T

∧ *methodsOf*(`TEnv`(*tid*)) (*m*) ≠ `undef` for *tid*=T1, T2, ..., Tk, T,

then *assigns*(T1, *m*)) ⊆ *assigns*(T, *m*)).

We start calculating from the induction hypothesis.

assigns(T1, *m*) ⊆ *assigns*(Tk, *m*)

⇒ < definition of *assigns*(Tk, *m*) of Figure 5.4 >

assigns(T1, *m*) ⊆ *assigns*(Tk, *m*)

∧ *assigns*(Tk, *m*) = *assignsOf*(*lookupMethod*(Tk, *m*)) ∩ *assigns*(*superOf*(Tk), *m*)

⇒ < definition of set containment and intersection >

assigns(T1, *m*) ⊆ *assigns*(Tk, *m*) ∧ *assigns*(Tk, *m*) ⊆ *assigns*(*superOf*(Tk), *m*)

⇔ < *superOf*(... *superOf*(Tk)...) = T from the definition of *superOf* in Figure 4.9 and *assigns* in Figure 4.11 and from the premise of the lemma, i.e., that *m* is declared in T >

assigns(T1, *m*) ⊆ *assigns*(Tk, *m*) ∧ *assigns*(Tk, *m*) ⊆ *assigns*(T, *m*)

⇒ < transitivity of set containment >

assigns(T1, *m*) ⊆ *assigns*(T, *m*)

■

Lemma 5.5 (Method Overriding): Let T1, T ∈ *TypeId* be classes allowed by our technique.

If T1 < T ∧ T.*m* ∈ *allMethodsIn*(*superOf*(T1)) ∧ ! *isOverridden*(T1, *m*)

then *noAddSideEffects*(T1, T, *m*) ∧ *validInvariant*(T1, T, *m*)

Proof:

T1 < T ∧ T.*m* ∈ *allMethodsIn*(*superOf*(T1)) ∧ ! *isOverridden*(T1, *m*)

⇒ < since class T1 is allowed by our rules, T1 must satisfy *requiredOverrides* of Figure 5.7 >

requiredOverrides(T1) ∧ T.*m* ∈ *allMethodsIn*(*superOf*(T1)) ∧ ! *isOverridden*(T1, *m*)

⇒ < definition of *requiredOverrides* in Figure 5.7 >

$(\forall\ U.n \in allMethodsIn(superOf(T1))\ :\ isOverridden(T1, n)$
$\quad \vee\ (\ noAddSideEffects(T1, U, n)\ \wedge\ validInvariant(T1, U, n)\ )\ )$
$\wedge\ T.m \in allMethodsIn(superOf(T1))\ \wedge\ !\ isOverridden(T1, m)$
$\Rightarrow <$ since $T.m \in allMethodsIn(superOf(T1))\ >$
$(\ isOverridden(T1, m)$
$\quad \vee\ (\ noAddSideEffects(T1, T, m)\ \wedge\ validInvariant(T1, T, m)\ )\ )$
$\wedge\ !\ isOverridden(T1, m)$
$\Rightarrow <$ logic $>$
$noAddSideEffects(T1, T, m)\ \wedge\ validInvariant(T1, T, m)$
■

5.2.2 Properties of the Program State

The lemmas in this subsection prove properties relating to expression evaluation, locations, and updating the program state; these lemmas will be used to simplify the validity proofs, given in subsection 5.2.6, for the axioms and inference rules of our verification logic. We also need to define the operator ≡ that will be used in the proofs given in this and later sections. The notation *rcvr* ≡ *e* means that *rcvr* and *e* are textually the same expressions; thus !(*rcvr* ≡ *e*) means *rcvr* and *e* are textually distinct expressions.

5.2.2.1 State Update Lemma

Our first lemma proves two properties about the program state after a location has been updated with a new value.

Lemma 5.6 (State Update): Let loc1, loc2 ∈ *Locations*, v ∈ *Value*, and S ∈ *State*.

If loc1 ≠ loc2, **then**

1. *getValue*(S[loc1 := v], loc1) = v
2. *getValue*(S[loc1 := v], loc2) = *getValue*(S, loc2)

Proof:

Part 1:

Case 1: Suppose loc1 is a local variable, i.e., loc1 = *loc*(x, `local`)

getValue(S[*loc*(x, `local`) := v], *loc*(x, `local`))
⇔ < update the stack by defn. of S[L := V] Figure 4.15 since *objRef*(*loc*(x, `local`)) = `local` >
getValue(*state*(*stackOf*(S)[*loc*(x, `local`) := v], *heapOf*(S)), *loc*(x, `local`))
⇔ < lookup in the stack by defn. of *getValue* Figure 4.15 since
objRef(*loc*(x, `local`)) = `local` >
stackOf(*state*(*stackOf*(S)[*loc*(x, `local`) := v], *heapOf*(S))) (*loc*(x, `local`))
⇔ < definition of *stackOf* in Figure 4.15 >
stackOf(S)[*loc*(x, `local`) := v] (*loc*(x, `local`))
⇔ < definition of OS[L := V] in Figure 4.15 >

(λ loc . **if** loc=*loc*(x, local) **then** v **else** *stackOf*(S)(loc)) (*loc*(x, local))

⇔ < function application >

v

Case 2: Suppose loc1 is a heap variable, i.e., loc1 = *loc*(x, r) with r ≠ local

getValue(S[*loc*(x, r) := v], *loc*(x, r))

⇔ < update the heap by defn. of S[L := V] Figure 4.15 since *objRef*(*loc*(x, r)) = r ≠ local >

getValue(*state*(*stackOf*(S), *heapOf*(S)[*loc*(x, r) := v]), *loc*(x, r))

⇔ < lookup in the heap by defn. of *getValue* Figure 4.15 since *objRef*(*loc*(x, r)) = r ≠ local >

heapOf(*state*(*stackOf*(S), *heapOf*(S)[*loc*(x, r) := v])) (*loc*(x, r)

⇔ < definition of *heapOf* in Figure 4.15 >

heapOf(S)[*loc*(x, r) := v] (*loc*(x, r))

⇔ < definition of OS[L := V] in Figure 4.15 >

(λ loc . **if** loc=*loc*(x, r) **then** v **else** *heapOf*(S)(loc)) (*loc*(x, r))

⇔ < function application >

v

Part 2:

Case 1: loc1 = *loc*(x, local) ∧ loc2 = *loc*(y, local), x ≠ y

getValue(S[*loc*(x, local) := v], *loc*(y, local))

⇔ < update the stack by defn. of S[L := V] Figure 4.15 since *objRef*(*loc*(x, local)) = local >

getValue(*state*(*stackOf*(S)[*loc*(x, local) := v], *heapOf*(S)), *loc*(y, local))

⇔ < lookup in the stack by defn. of *getValue* Figure 4.15 since

objRef(*loc*(y, local)) = local >

stackOf(*state*(*stackOf*(S)[*loc*(x, local) := v], *heapOf*(S))) (*loc*(y, local))

⇔ < definition of *stackOf* in Figure 4.15 >

stackOf(S)[*loc*(x, local) := v] (*loc*(y, local))

⇔ < definition of OS[L := V] in Figure 4.15 >

(λ loc . **if** loc=*loc*(x, local) **then** v **else** *stackOf*(S)(loc)) (*loc*(y, local))

⇔ < function application and x ≠ y ⇒ loc1 ≠ loc2 >

stackOf(S) (*loc*(y, local))

⇔ < definition of *getValue* in Figure 4.15 and *objRef*(*loc*(y, local)) = local >

getValue(S, *loc*(y, local))

Case 2: loc1 = *loc*(x, local), loc2 = *loc*(y, r), r ≠ local

getValue(S[*loc*(x, local) := v], *loc*(y, r))

⇔ < update the stack by defn. of S[L := V] Figure 4.15 since *objRef*(*loc*(x, local)) = local >

getValue(*state*(*stackOf*(S)[*loc*(x, local) := v], *heapOf*(S)), *loc*(y, r))

⇔ < lookup in the heap by defn. of *getValue* Figure 4.15 since *objRef*(*loc*(y, r)) = r ≠ local >

heapOf(*state*(*stackOf*(S)[*loc*(x, local) := v], *heapOf*(S))) (*loc*(y, r))

⇔ < definition of *heapOf* in Figure 4.15 >

heapOf(S) (*loc*(y, r))

⇔ < definition of *getValue* in Figure 4.15 and *objRef*(*loc*(y, r)) = r ≠ local >

getValue(S, *loc*(y, r))

Case 3: loc1 = *loc*(x, r), loc2 = *loc*(y, local), r ≠ local

getValue(S[*loc*(x, r) := v], *loc*(y, local))

⇔ < update the heap by defn. of S[L := V] Figure 4.15 since *objRef*(*loc*(x, r)) = r ≠ local >

getValue(*state*(*stackOf*(S), *heapOf*(S)[*loc*(x, r) := v]), *loc*(y, local))

⇔ < lookup in the stack by defn. of *getValue* Figure 4.15 since

objRef(*loc*(y, local)) = local >

stackOf(*state*(*stackOf*(S), *heapOf*(S)[*loc*(x, r) := v])) (*loc*(y, local)

⇔ < definition of *stackOf* in Figure 4.15 >

stackOf(S) (*loc*(y, local))

⇔ < definition of *getValue* in Figure 4.15 and *objRef*(*loc*(y, local)) = local >

getValue(S, *loc*(y, local))

Case 4: loc1 = *loc*(x, r1), loc2 = *loc*(y, r2), (r1 ≠ r2 ∨ x ≠ y) ∧ r1 ≠ local ∧ r2 ≠ local

getValue(S[*loc*(x, r1) := v], *loc*(y, r2))

⇔ < update the heap by defn. of S[L := V] Figure 4.15 since *objRef*(*loc*(x, r1)) = r1 ≠ local >

getValue(*state*(*stackOf*(S), *heapOf*(S)[*loc*(x, r1) := v]), *loc*(y, r2))

⇔ < lookup in the heap by defn. of *getValue* Figure 4.15 since *objRef*(*loc*(y, r2)) = r2 ≠ local >

heapOf(*state*(*stackOf*(S), *heapOf*(S)[*loc*(x, r1) := v])) (*loc*(y, r2)

⇔ < definition of *heapOf* in Figure 4.15 >

heapOf(S)[*loc*(x, r1) := v] (*loc*(y, r2))

⇔ < definition of OS[L := V] in Figure 4.15 >

(λ loc . **if** loc=*loc*(x, r1) **then** v **else** *heapOf*(S)(loc)) (*loc*(y, r2))

⇔ < function application and (r1 ≠ r2 ∨ x ≠ y) ⇔ loc1 ≠ loc2 >

heapOf(S) (*loc*(y, r2))

⇔ < definition of *getValue* in Figure 4.15 and *objRef*(*loc*(y, r2)) = r2 ≠ local >

getValue(S, *loc*(y, r2))

■

5.2.2.2 Variable references and locations

The next lemma says that if two variable references are textually different, then they denote different locations in the program state. Note, however, that this would not necessarily be true if our technique allowed direct assignment or access to fields of objects other than the receiver (see assumptions subsection 1.6.6). To ensure this property, the grammar of Java-C only allows assignment to local (stack) variables or fields of the receiver. Furthermore, for simplicity, we also assume that

variable names are not redefined in subclasses, i.e., `super.x` is not needed or in the grammar of Java-C.

Lemma 5.7 (Variable Reference):

If !(*vr* ≡ *vr2*) ∧ ([*vr*, S] ⇒$_{lv}$ vLoc) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc),
then vLoc ≠ v2Loc.

Proof:

Case 1: *vr* ≡ `this` ∧ (*vr2* ≡ `x` ∨ *vr2* ≡ `this.x` ∨ *vr2* ≡ `\result`)
([`this`, S] ⇒$_{lv}$ vLoc) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < L-Self rule of Figure 4.16 and definition of `thisLoc` of Figure 4.14 >
vLoc = *loc*(`this`, `local`) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < assumptions about *vr2* and L-VarId, L-Field, and L-Result rules of Figure 4.16 >
vLoc = *loc*(`this`, `local`) ∧ (v2Loc = *loc*(`x`, r) ∨ v2Loc = *loc*(`\result`, `global`))
⇒ < `this` is a reserved word so !(`this` ≡ `x`) ∧ !(`this` ≡ `\result`) >
vLoc ≠ v2Loc

Case 2: *vr* ≡ `\result` ∧ (*vr2* ≡ `x` ∨ *vr2* ≡ `this.x`)
([`\result`, S] ⇒$_{lv}$ vLoc) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < L-Result rule of Figure 4.16 and definition of `resultLoc` of Figure 4.14 >
vLoc = *loc*(`\result`, `global`) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < assumptions about *vr2* and the L-VarId and L-Field rules of Figure 4.14 >
vLoc = *loc*(`\result`, `global`) ∧ v2Loc = *loc*(`x`, r)
⇒ < `\result` ∉ *VarId* ⇒ !(`\result` ≡ `x`) >
vLoc ≠ v2Loc

Case 3: *vr* ≡ `x` ∧ !(`x` ≡ `y`) ∧ (*vr2* ≡ `y` ∨ *vr2* ≡ `this.y`)
([`x`, S] ⇒$_{lv}$ vLoc) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < L-VarId rule of Figure 4.14 >
vLoc = *loc*(`x`, `local`) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < assumption about *vr2* and the L-VarId and L-Field rules of Figure 4.14 >
vLoc = *loc*(`x`, `local`) ∧ v2Loc = *loc*(`y`, r)
⇒ < assumption that !(`x` ≡ `y`) >
vLoc ≠ v2Loc

Case 4: *vr* ≡ `x` ∧ *vr2* ≡ `this.x`
([`x`, S] ⇒$_{lv}$ vLoc) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < L-VarId rule of Figure 4.14 >
vLoc = *loc*(`x`, `local`) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)
⇒ < assumption that *vr2* ≡ `this.x` and the L-Field rule of Figure 4.14 >
vLoc = *loc*(`x`, `local`) ∧ r = *getValue*(S, `thisLoc`) ∧ v2Loc = *loc*(`x`, r)
⇒ < definition of `local` of Figure 4.14 >

vLoc = *loc*(x, *ref*(\Stack, *intV*(1))) ∧ r = *getValue*(S, thisLoc) ∧ v2Loc = *loc*(x, r)

⇒ < r references a heap object created by the new operator >

vLoc = *loc*(x, *ref*(\Stack, *intV*(1))) ∧ r = *getValue*(S, thisLoc) ∧ v2Loc = *loc*(x, r)

∧ T = *refType*(r) ∧ T ∈ *TypeId* ∧ \Stack = *refType*(vLoc) ∧ T = *refType*(v2Loc)

⇒ < \Stack ∉ *TypeId* ⇒ !(T ≡ \Stack) >

vLoc ≠ v2Loc

Case 5: *vr* ≡ this.x ∧ !(x ≡ y) ∧ *vr2* ≡ this.y

([this.x, S] ⇒$_{lv}$ vLoc) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)

⇒ < L-Field rule of Figure 4.14 >

vLoc = *loc*(x, r1) ∧ ([*vr2*, S] ⇒$_{lv}$ v2Loc)

⇒ < assumption about *vr2* and the L-Field rule of Figure 4.14 >

vLoc = *loc*(x, r1) ∧ v2Loc = *loc*(y, r2)

⇒ < by our assumption that !(x ≡ y) >

vLoc ≠ v2Loc

■

5.2.2.3 The Substitution Theorem

The Substitution Theorem proves an important property about expressions and the substitution of expressions for occurrences of variable references (including Java and JML pseudo variables such as this and \result). More specifically, it proves the relationship between substituting an expression for a variable and updating that variable's location with the value of the expression.

Theorem 5.8 (Substitution): Let S ∈ *State*, *P* and *e* be expressions, and *vr* be a variable or pseudovariable reference.

If ([*vr*, S] ⇒$_{lv}$ vLoc) ∧ ([*e*, S] ⇒$_{e}$ v),

then ([*P*[*vr*←*e*], S] ⇒$_{e}$ u) ∧ ([*P*, S[vLoc := v]] ⇒$_{e}$ u).

Proof:

The proof will be by induction on the structure of expression *P*.

Basis:

Case 1: Let *P* be a variable reference such that *P* = *vr*

([*vr*, S] ⇒$_{lv}$ vLoc) ∧ ([*e*, S] ⇒$_{e}$ v)

⇒ < E-VarRef rule of Figure 4.14 >

([*vr*, S[vLoc := v]] ⇒$_{e}$ *getValue*(S[vLoc := v], vLoc))

∧ ([*vr*, S] ⇒$_{lv}$ vLoc) ∧ ([*e*, S] ⇒$_{e}$ v)

⇒ < by the State Update Lemma (1) >

([*vr*, S[vLoc := v]] ⇒$_{e}$ v) ∧ ([*e*, S] ⇒$_{e}$ v)

⇒ < by definition of substitution, *vr*[*vr*←*e*] = *e* >

([*vr*, S[vLoc := v]] ⇒$_{e}$ v) ∧ ([*vr*[*vr*←*e*], S] ⇒$_{e}$ v)

Case 2: Let *P* be a variable reference such that *P* = *vr2*, *vr2* ≠ *vr*, and [*vr2*, S] $\Rightarrow_{lv}$ v2Loc

([*vr*, S] $\Rightarrow_{lv}$ vLoc) ∧ ([*vr2*, S] $\Rightarrow_{lv}$ v2Loc) ∧ ([*e*, S] $\Rightarrow_{e}$ v)

$\Rightarrow$ < by the Variable Reference Lemma above and our assumption *vr* ≠ *vr2* >

vLoc ≠ v2Loc ∧ ([*vr2*, S] $\Rightarrow_{lv}$ v2Loc) ∧ ([*e*, S] $\Rightarrow_{e}$ v)

$\Rightarrow$ < E-VarRef rule of Figure 4.14 >

([*vr2*, S[vLoc := v]] $\Rightarrow_{e}$ *getValue*(S[vLoc := v], v2Loc))

∧ vLoc ≠ v2Loc ∧ ([*vr2*, S] $\Rightarrow_{lv}$ v2Loc)

$\Rightarrow$ < State Update Lemma (2) and vLoc ≠ v2Loc >

([*vr2*, S[vLoc := v]] $\Rightarrow_{e}$ *getValue*(S, v2Loc)) ∧ ([*vr2*, S] $\Rightarrow_{lv}$ v2Loc)

$\Rightarrow$ < definition of substitution and *vr2* ≠ *vr* $\Rightarrow$ *vr2*[*vr*←*e*] = *vr2* >

([*vr2*, S[vLoc := v]] $\Rightarrow_{e}$ *getValue*(S, v2Loc)) ∧ ([*vr2*[*vr*←*e*], S] $\Rightarrow_{lv}$ v2Loc)

$\Rightarrow$ < E-VarRef rule of Figure 4.14 >

([*vr2*, S[vLoc := v]] $\Rightarrow_{e}$ *getValue*(S, v2Loc)) ∧ ([*vr2*[*vr*←*e*], S] $\Rightarrow_{e}$ *getValue*(S, v2Loc))

Case 3: *P* = *lit*

([*lit*, S] $\Rightarrow_{e}$ v1) ∧ ([*lit*, S[vLoc := v]] $\Rightarrow_{e}$ v2)

$\Rightarrow$ < E-Literal axiom of Figure 4.14 >

([*lit*, S] $\Rightarrow_{e}$ *mkVal*(*lit*)) ∧ ([*lit*, S[vLoc := v]] $\Rightarrow_{e}$ *mkVal*(*lit*))

$\Rightarrow$ < definition of substitution $\Rightarrow$ *lit*[*vr*←*e*] = *lit* >

([*lit*[*vr*←*e*], S] $\Rightarrow_{e}$ *mkVal*(*lit*)) ∧ ([*lit*, S[vLoc := v]] $\Rightarrow_{e}$ *mkVal*(*lit*))

Case 4: *P* = `this`

Follows from Cases 1 and 2 above, i.e., depending on whether or not *vr* = `this`.

Case 5: *P* = `( T ) null`

([`(T) null`, S] $\Rightarrow_{e}$ v1) ∧ ([`(T) null`, S[vLoc := v]] $\Rightarrow_{e}$ v2)

$\Rightarrow$ < E-CastNull axiom of Figure 4.14 >

([`(T) null`, S] $\Rightarrow_{e}$ *voidV*(null)) ∧ ([`(T) null`, S[vLoc := v]] $\Rightarrow_{e}$ *voidV*(null))

$\Rightarrow$ < definition of substitution $\Rightarrow$ `(T) null`[*vr*←*e*] = `(T) null` >

([`(T) null`[*vr*←*e*], S] $\Rightarrow_{e}$ *voidV*(null)) ∧ ([`(T) null`, S[vLoc := v]] $\Rightarrow_{e}$ *voidV*(null))

Induction Step:

The induction hypothesis assumes that the Substitution Lemma holds for all subexpressions of a larger expression. We will start our calculations from this hypothesis.

Case 1: *P* = `(` *e1* `)`

([*e1*, S[vLoc := v]] $\Rightarrow_{e}$ u) ∧ ([*e1*[*vr*←*e*], S] $\Rightarrow_{e}$ u)

$\Rightarrow$ < the E-Paren rule of Figure 4.14 >

([`(` *e1* `)`, S[vLoc := v]] $\Rightarrow_{e}$ u) ∧ ([`(` *e1*[*vr*←*e*] `)`, S] $\Rightarrow_{e}$ u)

$\Rightarrow$ < definition of substitution $\Rightarrow$ (*e1*)[*vr*←*e*] = (*e1*[*vr*←*e*]) >

([`(` *e1* `)`, S[vLoc := v]] $\Rightarrow_{e}$ u) ∧ ([`(` *e1* `)`[*vr*←*e*], S] $\Rightarrow_{e}$ u)

Case 2: *P = e1 bop e2*

([*e1*, S[vLoc := v]] $\Rightarrow_e$ u1) $\wedge$ ([*e2*, S[vLoc := v]] $\Rightarrow_e$ u2)

$\wedge$ ([*e1*[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ u1) $\wedge$ ([*e2*[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ u2)

$\Rightarrow$ < E-BinOp rule of Figure 4.14 >

([*e1 bop e2*, S[vLoc := v]] $\Rightarrow_e$ *apply*(*bop*, u1, u2))

$\wedge$ ([(*e1*[*vr*$\leftarrow$*e*] *bop e2*[*vr*$\leftarrow$*e*]), S] $\Rightarrow_e$ *apply*(*bop*, u1, u2))

$\Rightarrow$ < definition of substitution $\Rightarrow$ (*e1 bop e2*)[*vr*$\leftarrow$*e*] = *e1*[*vr*$\leftarrow$*e*] *bop e2*[*vr*$\leftarrow$*e*] >

([*e1 bop e2*, S[vLoc := v]] $\Rightarrow_e$ *apply*(*bop*, u1, u2))

$\wedge$ ([(*e1 bop e2*) [*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ *apply*(*bop*, u1, u2))

Case 3: *P = uop e1*

([*e1*, S[vLoc := v]] $\Rightarrow_e$ u) $\wedge$ ([*e1*[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ u)

$\Rightarrow$ < the E-UnOp rule of Figure 4.14 >

([*uop e1*, S[vLoc := v]] $\Rightarrow_e$ *apply*(*uop*, u)) $\wedge$ ([*uop e1*[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ *apply*(*uop*, u))

$\Rightarrow$ < definition of substitution $\Rightarrow$ (*uop e1*)[*vr*$\leftarrow$*e*] = *uop e1*[*vr*$\leftarrow$*e*] >

([*uop e1*, S[vLoc := v]] $\Rightarrow_e$ *apply*(*uop*, u)) $\wedge$ ([(*uop e1*)[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ *apply*(*uop*, u))

Case 4: *P* = `(T)` *e1*

([*e1*, S[vLoc := v]] $\Rightarrow_e$ u) $\wedge$ ([*e1*[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ u)

$\Rightarrow$ < (the E-Cast rule of Figure 4.14 >

([`(T)` *e1*, S[vLoc := v]] $\Rightarrow_e$ u) $\wedge$ ([`(T)` *e1*[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ u)

$\Rightarrow$ < definition of substitution $\Rightarrow$ (`(T)` *e1*) [*vr*$\leftarrow$*e*] = `(T)` *e1*[*vr*$\leftarrow$*e*] >

([`(T)` *e1*, S[vLoc := v]] $\Rightarrow_e$ u) $\wedge$ ([(`(T)` *e1*)[*vr*$\leftarrow$*e*], S] $\Rightarrow_e$ u)

■

To simplify the notation in some of our proofs we define the shorthand S<*e*> for evaluating expression *e* in a program state S.

Definition: **if** [*e*,S] $\Rightarrow_e$ v, **then** S<*e*> = v.

Using this definition, we can rewrite the Substitution Theorem 5.8 as follows:

Theorem 5.9 (Substitution): Let S $\in$ *State*, *P* and *e* be expressions, and *vr* be a variable or pseudovariable.

If ([*vr*, S] $\Rightarrow_{lv}$ vLoc) $\wedge$ S<*e*> = v,

then S<*P*[vr$\leftarrow$*e*]> = S[vLoc := v]<*P*>.

5.2.3 The Valid Invariant Theorem

In this subsection, we prove that, in our technique, super-calls can establish the invariant of the run-time type of the receiver just by establishing the invariant of the static type of the receiver. We first define some terminology to simplify the explanation of the approach used in our proofs. We define a *call chain* to be a sequence (nesting) of method calls in which each method in the chain directly calls the next method in the chain. We will represent such chains as sequences such as <C1,C2,C3,...,Ck>.

Thus C1 directly calls C2 and C2 directly calls C3, etc. An *object-call segment* is an object-call followed by a sequence of 0 or more self-calls and super-calls. When it is clear, we will refer to these sequences of calls as simply chains or segments. A *valid object-call segment* is a segment that is allowed by our rules. Similarly, a *valid call chain* is a chain allowed by our rules.

Lemma 5.10 (Call Chain): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T.

If method T.*m* can be executed by a program allowed by our technique,

then there is a valid call chain ending in T.*m* and

there must be a valid object-call segment that is a suffix of that chain.

Proof:

If a method T.*m* can be executed by a program allowed by our technique, then there must be a valid call chain ending in T.*m* since each call would have been checked by the T-rules given in Figures 5.1-5.3. So by definition, the call chain is valid.

Let < ... U1.*n1*, ... , Uk.*nk*, T.*m*> (where 0 ≤ *k*) be a valid call chain allowed by the current program execution. Starting at the end of the chain, move backward until the first object-call is located, say U1.*n1*; from that object-call to end of the chain is a valid object-call segment that ends with the call of T.*m*. ■

The next lemma says that, in our technique, a pivot field declared in the receiver cannot refer back to the receiver object containing that pivot field.

Lemma 5.11 (Self Pivot Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let *vr* be a variable accessible in method U.*n*. Let S ∈ *State* be an intermediate state at any point during the execution of U.*n*.

If *isPivot*(*vr*, T), **then** S<*vr* != `this`>.

Proof:

The only pivot fields accessible in a method are those declared in the receiver because the syntax of Java-C (Figure 4.5) only allows variables to reference fields of the receiver, local variables, and parameters; this is also one of our assumptions given in subsection 1.6.6.

Furthermore, it is not possible for an object X to have a field that owns object Y and for a field of Y to own X since one of the objects must be created first, i.e., it is not possible for an object created later to have a field that owns an object created earlier since it is not possible to write such code (see the T-rules in Figure 5.2). In particular, the right side of an assignment to a pivot field must be either a new object constructor call (T-New rule of Figure 5.2) or `null` (T-Null rule in Figure 5.2). All other assignments are not allowed because the predicate *okToAssign* of Figure 5.4 is an antecedent in the other rules in Figure 5.2. Therefore, this ownership relationship among objects, if considered as a directed graph, is non-cyclic. Specifically as required for this lemma, it is not possible for a pivot field

to reference the object containing that field since a non-null pivot field must reference an object created later through a new object constructor call (T-New rule of Figure 5.2). ■

The next few lemmas 5.12 through 5.24 prove various aliasing properties among owner variables; these lemmas are then used to prove that no two owner variables in the same context can be aliases when there are side-effects.

Lemma 5.12 (Owner Variable Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let *vr1* and *vr2* be variable references accessible at that same point. Let S ∈ *State* be an intermediate state also at the same point during the execution of U.*n*.

If (*isPivot*(*vr1*, U) ∨ *vr1* ∈ O) ∧ *vr2* ∈ O ∧ !(*vr1* ≡ *vr2*),
then S<*vr1* != *vr2*>.
Proof:
Suppose *vr2* ∈ O and !(*vr1* ≡ *vr2*), then there are two cases.
Case 1: *isPivot*(*vr1*, T)

In our technique, since *vr1* is a pivot field, *vr1* must be the first variable to contain a reference to a newly created object or it must be `null`; this is enforced by requiring that the right side of assignments to pivot fields be a new object constructor call or `null` (the T-New and T-Null rules allow these assignments since they are the only rules in Figure 5.2 that do have the *okToAssign* predicate of Figure 5.4 as an antecedent). Also, *vr2* must be the first variable to reference a newly created object because the only way a variable can be added to the set O of temporary owner variables is through an assignment with a new object constructor call as the right hand side (see the T-New rule of Figure 5.2); also, all other assignments remove *vr2* from O, i.e., *vr1* cannot have been assigned to *vr2* since *vr2* is a member of O. Furthermore, *vr2* cannot be assigned to *vr1* because the predicate *okToAssign* is an the antecedent of the T-ExpAssign rule. Thus *vr2* cannot be `null` and *vr1* and *vr2* cannot reference the same object when !(*vr1* ≡ *vr2*) since two different variables cannot both be the first to contain a reference to the same new object. Hence, S<*vr1* != *vr2*>.

Case 2: *vr1* ∈ O

In our technique, both *vr1* and *vr2* must be the first variable to contain a reference to a newly created object (because the T-New rule of Figure 5.2 is the only rule that adds variables to O). Also, the objects referenced by variables in O have been created during the current execution of T.*m*, so clearly *vr1* cannot reference the same object as *vr2* when !(*vr1* ≡ *vr2*), i.e., only one variable can be the first to contain a reference to a new object. All other assignments remove the target variable from O. Thus both *vr1* and *vr2* must reference different newly created objects, so S<*vr1* != *vr2*>.

■

Lemma 5.13 (Pivot Fields Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let *vr1* and *vr2* be variable references accessible in method U.*n*. Let S ∈ *State* be an intermediate state at an arbitrary point during the execution of U.*n*.

If *isPivot*(*vr1*, U) ∧ *isPivot*(*vr2*, U) ∧ !(*vr1* ≡ *vr2*) ∧ (S<*vr1* != `null`> ∨ S<*vr2* != `null`>), **then** S<*vr1* != *vr2*>.

Proof:

In our technique, both of the pivot fields *vr1* and *vr2* must be the first variable to contain a reference to a newly created object or be `null`; this is enforced by requiring that the right side of an assignment to a pivot field be a new object constructor call or `null` (see the T-New and T-Null rules in Figure 5.2). Furthermore, *vr1* cannot be assigned to *vr2* (and vice versa) by the predicate *okToAssign* in the antecedent of the T-ExpAssign rule; thus *vr1* and *vr2* cannot reference the same object during the execution of U.*n* when !(*vr1* ≡ *vr2*) (two different pivot field names) since only one field can be the first to contain a reference to a newly created object. Therefore, if either *vr1* or *vr2* references an object, then S<*vr1* != *vr2*>. ■

Lemma 5.14 (Self New Object Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let *vr* be a variable accessible at that same point. Let S ∈ *State* be an intermediate state also at the same point during the execution of U.*n*.

If *vr* ∈ O, **then** S<*vr* != `this`>.

Proof:

Since the receiver `this` cannot be the target of an assignment statement, `this` must reference an object that existed in the pre-state. However, the objects referenced by variables in O are created during the current execution of U.*n* (see the T-New rule of Figure 5.2, the only rule that adds new variables to O), so clearly *vr* cannot reference an object that existed prior to the execution of U.*n*. Therefore, S<*vr* != `this`>. ■

Lemma 5.15 (Parameter New Object Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let *vr* be a variable accessible at that same point. Let S ∈ *State* be an intermediate state also at the same point during the execution of U.*n*.

If *vr* ∈ O ∧ !(*vr* ≡ `p`) ∧ *typeOf*(`p`) ∈ *TypeId*,

then S<*vr* != `p`>.

Proof:

Suppose *vr* ∈ O and !(*vr* ≡ `p`), then there are two cases.

Case 1: `p` ∉ O

Since p ∉ O, either p has not been the target of an assignment statement (see the T-New rule of Figure 5.2) or null has been assigned to p (see the T-Null rule of Figure 5.2). No other assignments are allowed to p (see the T-CallAssign, T-SupCallAssign, and T-ExpAssign rules and, in particular, the predicate *okToAssign*). Therefore, p must still reference an object that existed in the pre-state or it contains null. The objects referenced by variables in O are created during the current execution of U.*n*, so clearly *vr* cannot reference an object that existed prior to the execution of U.*n* and it cannot be null since that would remove *vr* from O (see the T-Null rule of Figure 5.2). Therefore, *vr* and p cannot reference the same object and S<*vr* != null> so S<*vr* != p>.

Case 2: p ∈ O

If p ∈ O, then p has been the target of an assignment statement and the right side was a new object constructor call (see the T-New rule of Figure 5.2). Therefore, S<*vr* != p> by the Owner Variable Aliasing Lemma 5.12 since !(*vr* ≡ p).

■

Lemma 5.16 (New Pivots Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let *vr1* and *vr2* be variable references accessible in method U.*n*. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let S ∈ *State* be an intermediate state at that same point during the execution of U.*n*.

For all *pvt1* ∈ *allFieldsIn*(*typeOf*(*vr1*)) ∧ *pvt2* ∈ *allFieldsIn*(*typeOf*(*vr2*)),
If *vr1* ∈ O ∧ *vr2* ∈ O ∧ !(*vr1* ≡ *vr2*)
then (!*isPivot*(this.*pvt1*, *typeOf*(*vr1*)) ∨ S<*vr2* != *vr1*.*pvt1*>)
∧ (!*isPivot*(this.*pvt2*, *typeOf*(*vr2*)) ∨ S<*vr1* != *vr2*.*pvt2*>).

Proof:

Since *vr1* ∈ O and *vr2* ∈ O, S<*vr1* != this> and S<*vr2* != this> by the Self New Object Aliasing Lemma 5.14. Also, assignments to fields of objects other than the reciever are not allowed (see the syntax of assignment statements in Figure 4.5 and the T-rules and in particular the T-New rule of Figure 5.2) so fields of objects other than the receiver cannot be in O, i.e., *vr1* and *vr2* have to be fields of the receiver or local variables. Therefore, *vr1* and *vr2* cannot be pivot fields of each other since S<*vr1* != this> and S<*vr2* != this>, i.e., S<*vr2* != *vr1*.*pvt1*> and S<*vr1* != *vr2*.*pvt1*>. ■

Lemma 5.17 (Parameter New Pivots Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let *vr* be a variable accessible at that same point. Let S ∈ *State* be an intermediate state also at the same point during the execution of U.*n*.

For all *pPvt* ∈ *allFieldsIn*(*typeOf*(p)) ∧ *vPvt* ∈ *allFieldsIn*(*typeOf*(*vr*)),
If *vr* ∈ O ∧ !(*vr* ≡ p) ∧ *typeOf*(p) ∈ *TypeId* ∧ S<p != null>
∧ *isPivot*(this.*pPvt*, *typeOf*(p)) ∧ *isPivot*(this.*vPvt*, *typeOf*(*vr*)),
then (!*isPivot*(this.*pPvt*, *typeOf*(p)) ∨ S<*vr* != p.*pPvt*>)

$\wedge$ (!*isPivot*(`this.vPvt`, *typeOf*(*vr*)) $\vee$ S<`p != vr.vPvt`>).

Proof:

Suppose *vr* $\in$ O, !(*vr1* $\equiv$ `p`), and *typeOf*(`p`) $\in$ *TypeId*, then there are two cases.

Case 1: `p` $\notin$ O $\wedge$ S<`p != null`>

vr $\in$ O $\wedge$ `p` $\notin$ O $\wedge$ S<`p != null`> $\wedge$ *typeOf*(`p`) $\in$ *TypeId*

$\Rightarrow$ < if *isPivot*(`this.vPvt`, *typeOf*(*vr*)), then S<`p != vr.vPvt`> since `p` has not been the target of an assignment during the current execution of U.*n*, so `p` must reference an object that existed in the pre-state; therefore, `p` cannot reference a pivot object of the new object referenced by *vr* since *vr* was created after the object referenced by `p`, i.e., pivot fields must reference objects created after the containing object >

vr $\in$ O $\wedge$ `p` $\notin$ O $\wedge$ S<`p != null`> $\wedge$ *typeOf*(`p`) $\in$ *TypeId*

$\wedge$ (!*isPivot*(`this.vPvt`, *typeOf*(*vr*)) $\vee$ S<`p != vr.vPvt`>)

$\Rightarrow$ < if *isPivot*(`this.pPvt`, *typeOf*(`p`)), then S<*vr* != `p.pPvt`> since *vr* cannot be an alias of a pivot field of `p` because fields of objects other than the receiver `this` cannot be the target of an assignment, so `p.pPvt` $\notin$ O, i.e., `p.pPvt` references an object created by a method of object `p` in a different context >

(!*isPivot*(`this.pPvt`, *typeOf*(`p`)) $\vee$ S<*vr* != `p.pPvt`>)

$\wedge$ (!*isPivot*(`this.vPvt`, *typeOf*(*vr*)) $\vee$ S<`p != vr.vPvt`>)

Case 2: *vr* $\in$ O $\wedge$ `p` $\in$ O

vr $\in$ O $\wedge$ `p` $\in$ O $\wedge$ *typeOf*(`p`) $\in$ *TypeId*

$\wedge$ *isPivot*(`this.pPvt`, *typeOf*(`p`)) $\wedge$ *isPivot*(`this.vPvt`, *typeOf*(*vr*))

$\Rightarrow$ < by the New Pivots Aliasing Lemma 5.16 >

(!*isPivot*(`this.pPvt`, *typeOf*(`p`)) $\vee$ S<*vr* != `p.pPvt`>)

$\wedge$ (!*isPivot*(`this.vPvt`, *typeOf*(*vr*)) $\vee$ S<`p != vr.vPvt`>)

■

Lemma 5.18 (Self New Pivot Aliasing): Let U $\in$ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let *vr* be a variable accessible at that same point. Let S $\in$ *State* be an intermediate state also at the same point during the execution of U.*n*.

For all *vPvt* $\in$ *allFieldsIn*(*typeOf*(*vr*)),

If *vr* $\in$ O,

then (!*isPivot*(`this.vPvt`, *typeOf*(*vr*)) $\vee$ S<`this != vr.vPvt`>)

Proof:

The receiver `this` cannot be the target of an assignment, so `this` must reference an object that existed in the pre-state; thus `this` cannot reference a pivot object of the new object referenced by *vr* since pivot fields must reference objects created after the containing object. Also, `this` cannot be `null`, so S<`this != vr.vPvt`> when *vr*.*vPvt* is a pivot field. ■

Lemma 5.19 (Pivot New Pivot Aliasing): Let U ∈ *TypeId* be a valid class allowed by the rules of our technique and let *n* be a method declared in U. U.*n*. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in U.*n*. Let *vr1* and *vr2* be variables accessible at that same point. Let S ∈ *State* be an intermediate state also at the same point during the execution of U.*n*.

For all *pvt1* ∈ *allFieldsIn*(*typeOf*(*vr1*)) ∧ *pvt2* ∈ *allFieldsIn*(*typeOf*(*vr2*)),
If *vr1* ∈ O ∧ *isPivot*(*vr2*, U) ∧ S<*vr2* != `null`>,
then (!*isPivot*(`this`.*pvt1*, *typeOf*(*vr1*)) ∨ S<*vr2* != *vr1*.*pvt1*>)
∧ (!*isPivot*(`this`.*pvt2*, *typeOf*(*vr2*)) ∨ S<*vr1* != *vr2*.*pvt2*>)

Proof:

vr1 ∈ O ∧ *isPivot*(*vr2*, U) ∧ S<*vr2* != `null`>
⇒ < S<*vr1* != `this`> by the Self New Object Aliasing Lemma 5.14 >
S<*vr1* != `this`> ∧ *vr1* ∈ O ∧ *isPivot*(*vr2*, U)
⇒ < S<*vr2* != `this`> by the Self Pivot Aliasing Lemma 5.11 >
S<*vr1* != `this`> ∧ S<*vr2* != `this`> ∧ *vr1* ∈ O ∧ *isPivot*(*vr2*, U) ∧ S<*vr2* != `null`>
⇒ < if *isPivot*(`this`.*pvt1*, *typeOf*(*vr1*)), then S<*vr2* != *vr1*.*pvt1*> because
vr2 must be a pivot field of the receiver `this` (these are the only pivot fields accessible in the current context) and S<*vr1* != `this`>; furthermore, pivot fields of two different objects cannot reference the same object >
S<*vr2* != `this`> ∧ *vr1* ∈ O ∧ *isPivot*(*vr2*, U) ∧ S<*vr2* != `null`>
∧ (!*isPivot*(`this`.*pvt1*, *typeOf*(*vr1*)) ∨ S<*vr2* != *vr1*.*pvt1*>)
⇒ < if *isPivot*(`this`.*pvt2*, *typeOf*(*vr2*)), then S<*vr1* != *vr2*.*pvt2*> since *vr1* contains a reference to an object created during the current execution of U.*n*, whereas *vr2*.*pvt2* contains `null` or a reference to an object created by a method of *vr2* in a different context (i.e., S<*vr2* != `this`> so *vr2*.*pvt2* cannot be directly assigned in U.*n*); thus *vr1* and *vr2*.*pvt2* cannot reference the same new object since the assignments to both have to be done in two different contexts and, by the T-New rule of Figure 5.2, both have to have a new object constructor call as the right hand side >
(!*isPivot*(`this`.*pvt1*, *typeOf*(*vr1*)) ∨ S<*vr2* != *vr1*.*pvt1*>)
∧ (!*isPivot*(`this`.*pvt2*, *typeOf*(*vr2*)) ∨ S<*vr1* != *vr2*.*pvt2*>)

■

Lemma 5.20 (Valid Parameter): Let *rcvr*.*m*(*e*) be a method call allowed by our technique in some method U.*n*. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at the point of the call in U.*n*.

If *typeOf*(*e*) ∈ *TypeId*,
then (*rcvr* ∈ O ∨ *rcvr* ≡ `p` ∨ *rcvr* ≡ `this` ∨ *isPivot*(*rcvr*, U))

$\wedge$ ($e \in$ O $\vee$ $e \equiv$ `p` $\vee$ $e \equiv$ `this` $\vee$ *isPivot*(e, U)).

Proof:

Since *rcvr*`.`*m*(*e*) is allowed by our technique, it must satisfy the predicate *invariantOK* which is an antecedent of the T-Call and T-SupCall rules of Figure 5.1 and indirectly an antecedent of the T-CallAssign and T-SupCallAssign rules of Figure 5.2, so we start our calculation from that predicate.

invariantOK(*rcvr*, *e*, O, U) $\wedge$ *typeOf*(*e*) $\in$ *TypeId*

$\Rightarrow$ < definition of *invariantOK* of Figure 5.4 >

isOwner(*rcvr*, O, U) $\wedge$ (*typeOf*(*e*) $\notin$ *TypeId* $\vee$ *isOwner*(*e*, O, U)) $\wedge$ *typeOf*(*e*) $\in$ *TypeId*

$\Rightarrow$ < logic >

isOwner(*rcvr*, O, U) $\wedge$ *isOwner*(*e*, O, U)

$\Rightarrow$ < by the definition of *isOwner* of Figure 5.65.5 >

(*rcvr* $\in$ O $\vee$ *rcvr* $\equiv$ `p` $\vee$ *rcvr* $\equiv$ `this` $\vee$ *isPivot*(*rcvr*, U))

$\wedge$ (*e* $\in$ O $\vee$ *e* $\equiv$ `p` $\vee$ *e* $\equiv$ `this` $\vee$ *isPivot*(*e*, U))

■

Lemma 5.21 (Self Callback): Let *rcvr*`.`*m*(*e*) be a method call allowed by our technique in some method U`.`*n*. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at the point of the call in U`.`*n*. Let S $\in$ *State* be an intermediate state at that same point during the execution of U`.`*n*.

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) $\wedge$ S<*rcvr* != `this`> $\wedge$ S<*e* != `this`>

then *rcvr*`.`*m*(*e*) cannot callback to a method of the current receiver `this`.

Proof:

Clearly, a callback to a method of `this` cannot occur through *e* when *typeOf*(*e*) $\notin$ *TypeId*. So we assume that *typeOf*(*e*) $\in$ *TypeId* and thus, by the Valid Parameter Lemma 5.20, the actual parameters allowed by our technique in the call of *rcvr*`.`*m*(*e*) must satisfy the following.

(*rcvr* $\in$ O $\vee$ *rcvr* $\equiv$ `p` $\vee$ *rcvr* $\equiv$ `this` $\vee$ *isPivot*(*rcvr*, U))

$\wedge$ (*e* $\in$ O $\vee$ *e* $\equiv$ `p` $\vee$ *e* $\equiv$ `this` $\vee$ *isPivot*(*e*, U))

As explained in the Self Pivot Aliasing Lemma 5.11, the ownership relationship among pivot fields of objects, if considered as a directed graph, is non-cyclic. So a pivot object X cannot have a pivot field that refers back to the object containing the pivot field X. Hence, passing a pivot field as an argument cannot result in a callback to a method of the receiver, i.e., *isPivot*(*rcvr*, U) and *isPivot*(*e*, U) in the above assertion cannot lead to such callbacks. Also, *rcvr* $\in$ O and *e* $\in$ O cannot lead to a callback to a method of the receiver because *rcvr* and *e* cannot have pivot fields that reference the receiver since the receiver was created earlier (see also the Self New Pivot Aliasing Lemma 5.18). Thus the only possibilities left that are allowed by our technique are through the receiver and formal parameter of a call, but the premise of the lemma says that `this` is neither argument so no such callback would be allowed. ■

Lemma 5.22 (Self-Call Pivot): Let *rcvr*`.`*m*(*e*) be a method call allowed by our technique in some method U`.`*n*. Let S ∈ *State* be an intermediate state at the point just prior to this call during the execution of U`.`*n*.

For all *pvt* ∈ *allFieldsIn*(*typeOf*(*rcvr*)),

If *typeOf*(*e*) ∈ *TypeId* ∧ *isPivot*(`this.`*pvt*, *typeOf*(*rcvr*)) ∧ S<*rcvr* != `null`> ∧ S<*e* != *rcvr*`.`*pvt*>

then !(S<*rcvr* == `this`> ∧ S<*e* == `this.`*pvt*>)

Proof:

We will prove the contrapositive since that is quite easy.

S<*rcvr* == `this`> ∧ S<*e* == `this.`*pvt*>

⇒ < substituting *rcvr* for `this` since they are aliases in state S >

S<*e* == *rcvr*`.`*pvt*>

⇒ < logic >

typeOf(*e*) ∉ *TypeId* ∨ !*isPivot*(`this.`*pvt*, *typeOf*(*e*)) ∨ S<*rcvr* == `null`> ∨ S<*e* == *rcvr*`.`*pvt*>

■

Lemma 5.23 (This-Argument Call): Let *rcvr*`.`*m*(*e*) be a method call allowed by our technique in some method U`.`*n*. Let S ∈ *State* be an intermediate state at the point just prior to this call during the execution of U`.`*n*.

For all *pvt* ∈ *allFieldsIn*(*typeOf*(*e*)),

If *typeOf*(*e*) ∈ *TypeId* ∧ *isPivot*(`this.`*pvt*, *typeOf*(*e*)) ∧ S<*e* != `null`> ∧ S<*rcvr* != *e*`.`*pvt*>

then !(S<*e* == `this`> ∧ S<*rcvr* == `this.`*pvt*>)

Proof:

Again, we prove the contrapositive.

S<*e* == `this`> ∧ S<*rcvr* == `this.`*pvt*>

⇒ < substituting *e* for `this` since they are aliases in state S >

S<*rcvr* == *e*`.`*pvt*>

⇒ < logic >

typeOf(*e*) ∉ *TypeId* ∨ !*isPivot*(`this.`*pvt*, *typeOf*(*e*)) ∨ S<*e* == `null`> ∨ S<*rcvr* == *e*`.`*pvt*>

■

We do not want side-effects to an object referenced by a formal parameter to also modify the state of the receiver or vice versa because we do not want side-effects to one argument to possibly invalidate the other object's invariant. If the invariant of one of the arguments is invalidated, then subsequent calls on that object would lead to unverifiable behavior, since the invariant does not hold and, in our technique, the code may not be available. Therefore, when a method has side-effects, we have to make sure that argument objects do not share state; for example, we have to make sure that a formal parameter is not be an alias of one of the receiver's pivot objects. For the same reason, we have to

make sure that a formal parameter and the receiver are not aliases. The next lemma proves that the checking rules of our technique prevent these situations from occurring.

Lemma 5.24 (Actual Parameter Aliasing): Let *rcvr*.*m*(*e*) be a method call allowed by our technique in some method U.*n*. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at the point of this call in U.*n*. Let S ∈ *State* be an intermediate state at that same point during the execution of U.*n*.

For all *rPvt* ∈ *allFieldsIn*(*typeOf*(*rcvr*)) ∧ *pPvt* ∈ *allFieldsIn*(*typeOf*(*e*)),

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*)

∧ *typeOf*(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }

then S<*rcvr* != *e*> ∧ (!*isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*.*rPvt*>)

∧ (!*isPivot*(`this.pPvt`, *typeOf*(*e*)) ∨ S<*rcvr* != *e*.*pPvt*>)

Proof:

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*)

∧ *typeOf*(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }

⇒ < since *rcvr*.*m*(*e*) is allowed, this call must satisfy *aliasingOK* based on the T-Call and T-SupCall rules of Figure 5.1 >

typeOf(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }

∧ *aliasingOK*(*rcvr*, *e*, U, T, *m*)

⇒ < definition of *aliasingOK* of Figure 5.4 >

typeOf(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }

∧ (*assigns*(T, *m*) = { }

∨ (!(*rcvr* ≡ *e*) ∧ (!*isPivot*(*e*, U) ∨ !(*rcvr* ≡ `this`)) ∧ (!*isPivot*(*rcvr*, U) ∨ !(*e* ≡ `this`)))

⇒ < logic >

typeOf(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }

∧ !(*rcvr* ≡ *e*) ∧ (!*isPivot*(*e*, U) ∨ !(*rcvr* ≡ `this`)) ∧ (!*isPivot*(*rcvr*, U) ∨ !(*e* ≡ `this`)))

We would be finished at this point were it not for aliasing. However, in our technique, we do not prevent the receiver or pivot objects from being aliased through local variables or non-pivot fields. Therefore, we still need to prove that our technique prevents the aliasing (with respect to the program state) specified in the conclusion of this lemma. We will split the rest of the proof into cases as explained below; each case will continue from the above assertion.

By the Call Chain Lemma 5.10, there is a valid call chain ending with T.*m* since the call of *rcvr*.*m*(*e*) is allowed by our technique from U.*n* (i.e., we are executing in U.*n* when the call is made). Therefore, the proof will be by induction on the number of calls in a valid call chain prior to the call of *rcvr*.*m*(*e*).

Basis: *k* = 0, i.e., <T.*m*> so T.*m* is not called by method U.*n*.

Vacuously true since the lemma and induction hypothesis (and all of our rules) are in the context of a method U . *n* that calls method T . *m*. This is also sensible because, in our technique, the entry point of a Java program is a static method that would have no receiver or pivot fields that are accessible in that context.

Induction Step: $k = \mathsf{N}$.

The induction hypothesis asserts that

For all $rPvt \in allFieldsIn(typeOf(rcvr)) \wedge pPvt \in allFieldsIn(typeOf(e))$,

If $\mathrm{T} = whereMethodDecl(typeOf(rcvr), m)$

$\wedge\ typeOf(e) \in TypeId \wedge \mathsf{S}\langle rcvr\ \texttt{!= null}\rangle \wedge \mathsf{S}\langle e\ \texttt{!= null}\rangle \wedge assigns(\mathrm{T}, m) \neq \{\ \}$

$\wedge\ k < \mathsf{N} \wedge \langle \mathrm{T1}\ \texttt{.}\ m1, \mathrm{T2}\ \texttt{.}\ m2, \ldots, \mathrm{U}\ \texttt{.}\ n, \mathrm{T}\ \texttt{.}\ m\rangle$ is a valid call chain

with U . *n* the *k*th call in the chain,

then $\mathsf{S}\langle rcvr\ \texttt{!=}\ e\rangle \wedge (!isPivot(\texttt{this .}\ rPvt, typeOf(rcvr)) \vee \mathsf{S}\langle e\ \texttt{!=}\ rcvr\ \texttt{.}\ rPvt\rangle)$

$\wedge\ (!isPivot(\texttt{this .}\ pPvt, typeOf(e)) \vee \mathsf{S}\langle rcvr\ \texttt{!=}\ e\ \texttt{.}\ pPvt\rangle)$

From this hypothesis we can conclude that the following assertion holds at the start of the execution of U . *n* when $k = \mathsf{N}$ by substituting `this` for *rcvr* and `p` for *e*.

$\mathsf{S}\langle\texttt{this != p}\rangle \wedge (!isPivot(\texttt{this .}\ rPvt, typeOf(\texttt{this})) \vee \mathsf{S}\langle\texttt{p != this .}\ rPvt\rangle)$

$\wedge\ (!isPivot(\texttt{this .}\ pPvt, typeOf(\texttt{p})) \vee \mathsf{S}\langle\texttt{this != p .}\ pPvt\rangle)$

Since $typeOf(e) \in TypeId$ and $rcvr\ \texttt{.}\ m(e)$ is allowed by our technique, we have, by the Valid Parameter Lemma 5.20, the following valid combinations of actual parameters.

$(rcvr \in \mathrm{O} \vee rcvr \equiv \texttt{p} \vee rcvr \equiv \texttt{this} \vee isPivot(rcvr, \mathrm{U}))$

$\wedge\ (e \in \mathrm{O} \vee e \equiv \texttt{p} \vee e \equiv \texttt{this} \vee isPivot(e, \mathrm{U}))$

Therefore, we continue from where we left off by splitting the proof into the 16 valid combinations of *rcvr* and *e* as given in the above formula.

Case 1: $rcvr \in \mathrm{O} \wedge e \in \mathrm{O}$

$typeOf(e) \in TypeId \wedge \mathsf{S}\langle rcvr\ \texttt{!= null}\rangle \wedge \mathsf{S}\langle e\ \texttt{!= null}\rangle \wedge assigns(\mathrm{T}, m) \neq \{\ \}$

$\wedge\ !(rcvr \equiv e) \wedge (!isPivot(e, \mathrm{U}) \vee !(rcvr \equiv \texttt{this})) \wedge (!isPivot(rcvr, \mathrm{U}) \vee !(e \equiv \texttt{this}))$

$\wedge\ rcvr \in \mathrm{O} \wedge e \in \mathrm{O}$

$\Rightarrow$ < logic >

$!(rcvr \equiv e) \wedge rcvr \in \mathrm{O} \wedge e \in \mathrm{O}$

$\Rightarrow$ < $\mathsf{S}\langle rcvr\ \texttt{!=}\ e\rangle$ by the Owner Variable Aliasing Lemma 5.12 >

$\mathsf{S}\langle rcvr\ \texttt{!=}\ e\rangle \wedge rcvr \in \mathrm{O} \wedge e \in \mathrm{O}$

$\Rightarrow$ < *rcvr* and *e* cannot reference pivot objects of each other by the New Pivots Aliasing Lemma 5.16 >

$\mathsf{S}\langle rcvr\ \texttt{!=}\ e\rangle \wedge (!isPivot(\texttt{this .}\ rPvt, typeOf(rcvr)) \vee \mathsf{S}\langle e\ \texttt{!=}\ rcvr\ \texttt{.}\ rPvt\rangle)$

$\wedge\ (!isPivot(\texttt{this .}\ pPvt, typeOf(e)) \vee \mathsf{S}\langle rcvr\ \texttt{!=}\ e\ \texttt{.}\ pPvt\rangle)$

Case 2: $rcvr \in \mathrm{O} \wedge e \equiv \texttt{this}$

typeOf(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }
∧ !(*rcvr* ≡ *e*) ∧ (!*isPivot*(*e*, U) ∨ !(*rcvr* ≡ `this`)) ∧ (!*isPivot*(*rcvr*, U) ∨ !(*e* ≡ `this`))
∧ *rcvr* ∈ O ∧ *e* ≡ `this`

⇒ < logic >

(!(*e* ≡ `this`) ∨ !*isPivot*(*rcvr*, U)) ∧ *rcvr* ∈ O ∧ *e* ≡ `this`

⇒ < logic >

!*isPivot*(*rcvr*, U) ∧ *rcvr* ∈ O ∧ *e* ≡ `this`

⇒ < S<*rcvr* != `this`> by the Self New Object Aliasing Lemma 5.14 >

!*isPivot*(*rcvr*, U) ∧ *rcvr* ∈ O ∧ *e* ≡ `this` ∧ S<*rcvr* != `this`>

⇒ < by the Self New Pivot Aliasing Lemma 5.18 >

!*isPivot*(*rcvr*, U) ∧ *rcvr* ∈ O ∧ *e* ≡ `this`
∧ S<*rcvr* != `this`> ∧ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) ∨ S<`this` != *rcvr*`.`*rPvt*>)

⇒ < substituting *e* for `this` in the last two conjuncts >

!*isPivot*(*rcvr*, U) ∧ *rcvr* ∈ O ∧ *e* ≡ `this`
∧ S<*rcvr* != *e*> ∧ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*`.`*rPvt*>)

⇒ < if *isPivot*(`this.`*pPvt*, *typeOf*(`this`)), then !(*rcvr* ≡ `this.`*pPvt*) since !*isPivot*(*rcvr*, U);
so, by the Owner Variable Aliasing Lemma 5.12, S<*rcvr* != `this.`*pPvt*> >

rcvr ∈ O ∧ *e* ≡ `this`
∧ S<*rcvr* != *e*> ∧ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*`.`*rPvt*>)
∧ (!*isPivot*(`this.`*pPvt*, *typeOf*(`this`)) ∨ S<*rcvr* != `this.`*pPvt*>)

⇒ < logic and substituting *e* for `this` in the last conjunct >

S<*rcvr* != *e*> ∧ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*`.`*rPvt*>)
∧ (!*isPivot*(`this.`*pPvt*, *typeOf*(*e*)) ∨ S<*rcvr* != *e*`.`*pPvt*>)

Case 3: *rcvr* ∈ O ∧ *e* ≡ `p`

typeOf(*e*) ∈ *TypeId* ∧ S<*rcvr* != `null`> ∧ S<*e* != `null`> ∧ *assigns*(T, *m*) ≠ { }
∧ !(*rcvr* ≡ *e*) ∧ (!*isPivot*(*e*, U) ∨ !(*rcvr* ≡ `this`)) ∧ (!*isPivot*(*rcvr*, U) ∨ !(*e* ≡ `this`))
∧ *rcvr* ∈ O ∧ *e* ≡ `p`

⇒ < logic >

!(*rcvr* ≡ *e*) ∧ *rcvr* ∈ O ∧ *e* ≡ `p` ∧ *typeOf*(*e*) ∈ *TypeId* ∧ S<*e* != `null`>

⇒ < S<*rcvr* != *e*> by the Parameter New Object Aliasing Lemma 5.15 >

S<*rcvr* != *e*> ∧ *rcvr* ∈ O ∧ *e* ≡ `p` ∧ *typeOf*(*e*) ∈ *TypeId* ∧ S<*e* != `null`>

⇒ < by the Parameter New Pivots Aliasing Lemma 5.17 >

S<*rcvr* != *e*> ∧ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) ∨ S<`p` != *rcvr*`.`*rPvt*>)
∧ (!*isPivot*(`this.`*pPvt*, *typeOf*(`p`)) ∨ S<*rcvr* != `p.`*pPvt*>)

⇒ < substituting *e* for `p` >

S<*rcvr* != *e*> ∧ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*`.`*rPvt*>)
∧ (!*isPivot*(`this.`*pPvt*, *typeOf*(*e*)) ∨ S<*rcvr* != *e*`.`*pPvt*>)

Case 4: $rcvr \in O \wedge isPivot(e, U)$

$typeOf(e) \in TypeId \wedge S<rcvr \text{ != } \texttt{null}> \wedge S<e \text{ != } \texttt{null}> \wedge assigns(T, m) \neq \{\}$
$\wedge\ !(rcvr \equiv e) \wedge (!isPivot(e, U) \vee !(rcvr \equiv \texttt{this})) \wedge (!isPivot(rcvr, U) \vee !(e \equiv \texttt{this}))$
$\wedge\ rcvr \in O \wedge isPivot(e, U)$

$\Rightarrow <$ logic $>$

$!(rcvr \equiv e) \wedge rcvr \in O \wedge isPivot(e, U) \wedge S<e \text{ != } \texttt{null}>$

$\Rightarrow <$ $S<rcvr \text{ != } e>$ by the Owner Variable Aliasing Lemma 5.12 $>$

$S<rcvr \text{ != } e> \wedge rcvr \in O \wedge isPivot(e, U) \wedge S<e \text{ != } \texttt{null}>$

$\Rightarrow <$ by the Pivot New Pivot Aliasing Lemma 5.19 $>$

$S<rcvr \text{ != } e> \wedge (!isPivot(\texttt{this.}rPvt, typeOf(rcvr)) \vee S<e \text{ != } rcvr\texttt{.}rPvt>)$
$\wedge\ (!isPivot(\texttt{this.}pPvt, typeOf(e)) \vee S<rcvr \text{ != } e\texttt{.}pPvt>)$

Case 5: $rcvr \equiv \texttt{this} \wedge e \in O$

$typeOf(e) \in TypeId \wedge S<rcvr \text{ != } \texttt{null}> \wedge S<e \text{ != } \texttt{null}> \wedge assigns(T, m) \neq \{\}$
$\wedge\ !(rcvr \equiv e) \wedge (!isPivot(e, U) \vee !(rcvr \equiv \texttt{this})) \wedge (!isPivot(rcvr, U) \vee !(e \equiv \texttt{this}))$
$\wedge\ rcvr \equiv \texttt{this} \wedge e \in O$

$\Rightarrow <$ the proof is analogous to Case 2 $>$

$S<rcvr \text{ != } e> \wedge (!isPivot(\texttt{this.}rPvt, typeOf(rcvr)) \vee S<e \text{ != } rcvr\texttt{.}rPvt>)$
$\wedge\ (!isPivot(\texttt{this.}pPvt, typeOf(e)) \vee S<rcvr \text{ != } e\texttt{.}pPvt>)$

Case 6: $rcvr \equiv \texttt{this} \wedge e \equiv \texttt{this}$

$!(rcvr \equiv e) \wedge rcvr \equiv \texttt{this} \wedge e \equiv \texttt{this}$

$\Rightarrow <$ definition of $\equiv$ $>$

false

So this combination is not allowed by our technique.

Case 7: $rcvr \equiv \texttt{this} \wedge isPivot(e, U)$

$typeOf(e) \in TypeId \wedge S<rcvr \text{ != } \texttt{null}> \wedge S<e \text{ != } \texttt{null}> \wedge assigns(T, m) \neq \{\}$
$\wedge\ !(rcvr \equiv e) \wedge (!isPivot(e, U) \vee !(rcvr \equiv \texttt{this})) \wedge (!isPivot(rcvr, U) \vee !(e \equiv \texttt{this}))$
$\wedge\ rcvr \equiv \texttt{this} \wedge isPivot(e, U)$

$\Rightarrow <$ logic $>$

$!(rcvr \equiv e) \wedge (!(rcvr \equiv \texttt{this}) \vee !isPivot(e, U)) \wedge rcvr \equiv \texttt{this} \wedge isPivot(e, U)$

$\Rightarrow <$ logic $>$

$!(rcvr \equiv e) \wedge !isPivot(e, U) \wedge rcvr \equiv \texttt{this} \wedge isPivot(e, U)$

$\Rightarrow <$ logic $>$

false

So this combination is not allowed by our technique.

Case 8: $isPivot(rcvr, U) \wedge e \in O$

$typeOf(e) \in TypeId \wedge S<rcvr \text{ != } \texttt{null}> \wedge S<e \text{ != } \texttt{null}> \wedge assigns(T, m) \neq \{\}$
$\wedge\ !(rcvr \equiv e) \wedge (!isPivot(e, U) \vee !(rcvr \equiv \texttt{this})) \wedge (!isPivot(rcvr, U) \vee !(e \equiv \texttt{this}))$

$\wedge$ *isPivot*(*rcvr*, U) $\wedge$ $e \in$ O $\wedge$ S<*rcvr* != `null`>

$\Rightarrow$ < the proof is analogous to Case 4 >

S<*rcvr* != *e*> $\wedge$ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) $\vee$ S<*e* != *rcvr*`.`*rPvt*>)

$\wedge$ (!*isPivot*(`this.`*pPvt*, *typeOf*(*e*)) $\vee$ S<*rcvr* != *e*`.`*pPvt*>)

Case 9: *isPivot*(*rcvr*, U) $\wedge$ *e* $\equiv$ `this`

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`> $\wedge$ *assigns*(T, *m*) $\neq$ { }

$\wedge$!(*rcvr* $\equiv$ *e*) $\wedge$ (!*isPivot*(*e*, U) $\vee$!(*rcvr* $\equiv$ `this`)) $\wedge$ (!*isPivot*(*rcvr*, U) $\vee$!(*e* $\equiv$ `this`))

$\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *e* $\equiv$ `this`

$\Rightarrow$ < logic >

!(*rcvr* $\equiv$ *e*) $\wedge$ (!(*e* $\equiv$ `this`) $\vee$!*isPivot*(*rcvr*, U)) $\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *e* $\equiv$ `this`

$\Rightarrow$ < logic >

!(*rcvr* $\equiv$ *e*) $\wedge$!(*e* $\equiv$ `this`) $\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *e* $\equiv$ `this`

$\Rightarrow$ < logic >

false

So this combination is not allowed by our technique.

Case 10: *isPivot*(*rcvr*, U) $\wedge$ *isPivot*(*e*, U)

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`> $\wedge$ *assigns*(T, *m*) $\neq$ { }

$\wedge$!(*rcvr* $\equiv$ *e*) $\wedge$ (!*isPivot*(*e*, U) $\vee$!(*rcvr* $\equiv$ `this`)) $\wedge$ (!*isPivot*(*rcvr*, U) $\vee$!(*e* $\equiv$ `this`))

$\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *isPivot*(*e*, U)

$\Rightarrow$ < logic >

!(*rcvr* $\equiv$ *e*) $\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *isPivot*(*e*, U) $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`>

$\Rightarrow$ < S<*rcvr* != *e*> by the Pivot Fields Aliasing Lemma 5.13 >

S<*rcvr* != *e*> $\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *isPivot*(*e*, U) $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`>

$\Rightarrow$ < S<*rcvr* != `this`> and S<*e* != `this`> by the Self Pivot Aliasing Lemma 5.11 >

S<*rcvr* != *e*> $\wedge$ S< *rcvr* != `this`> $\wedge$ S<*e* != `this`>

$\wedge$ *isPivot*(*rcvr*, U) $\wedge$ *isPivot*(*e*, U) $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`>

$\Rightarrow$ < *rcvr* and *e* are pivot fields of `this` so they cannot also be pivot fields of each other since neither is the current receiver and pivot fields of different objects cannot reference the same object >

S<*rcvr* != *e*> $\wedge$ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) $\vee$ S<*e* != *rcvr*`.`*rPvt*>)

$\wedge$ (!*isPivot*(`this.`*pPvt*, *typeOf*(*e*)) $\vee$ S<*rcvr* != *e*`.`*pPvt*>)

Case 11: *rcvr* $\equiv$ `p` $\wedge$ *e* $\in$ O

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`> $\wedge$ *assigns*(T, *m*) $\neq$ { }

$\wedge$!(*rcvr* $\equiv$ *e*) $\wedge$ (!*isPivot*(*e*, U) $\vee$!(*rcvr* $\equiv$ `this`)) $\wedge$ (!*isPivot*(*rcvr*, U) $\vee$!(*e* $\equiv$ `this`))

$\wedge$ *rcvr* $\equiv$ `p` $\wedge$ *e* $\in$ O

$\Rightarrow$ < the proof is analogous to Case 3 >

S<*rcvr* != *e*> $\wedge$ (!*isPivot*(`this.`*rPvt*, *typeOf*(*rcvr*)) $\vee$ S<*e* != *rcvr*`.`*rPvt*>)

$\wedge$ (!*isPivot*(`this.pPvt`, *typeOf*(*e*)) $\vee$ S<*rcvr* != *e*`.pPvt`>)

Case 12: *rcvr* $\equiv$ `p` $\wedge$ *e* $\equiv$ `p`

!(*rcvr* $\equiv$ *e*) $\wedge$ *rcvr* $\equiv$ `p` $\wedge$ *e* $\equiv$ `p`

$\Rightarrow$ < definition of $\equiv$ >

false

So this combination is not allowed by our technique.

The last four cases need to use the induction hypothesis because they involve potential aliasing through the receiver and formal parameter, i.e., we have to consider the execution of U`.`*n* in the context of a valid call chain.

Case 13: *rcvr* $\equiv$ `this` $\wedge$ *e* $\equiv$ `p`

If `p` $\in$ O, then this case holds from Case 5 above. Therefore, we assume that `p` $\notin$ O and there have been no assignments to `p`.

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`> $\wedge$ *assigns*(T, *m*) $\neq$ { }

$\wedge$!(*rcvr* $\equiv$ *e*) $\wedge$ (!*isPivot*(*e*, U) $\vee$!(*rcvr* $\equiv$ `this`)) $\wedge$ (!*isPivot*(*rcvr*, U) $\vee$!(*e* $\equiv$ `this`))

$\wedge$ *isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) $\wedge$ *isPivot*(`this.pPvt`, *typeOf*(*e*))

$\wedge$ *rcvr* $\equiv$ `this` $\wedge$ *e* $\equiv$ `p`

$\Rightarrow$ < logic >

rcvr $\equiv$ `this` $\wedge$ *e* $\equiv$ `p` $\wedge$ S<*e* != `null`>

$\wedge$ *isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) $\wedge$ *isPivot*(`this.pPvt`, *typeOf*(*e*))

$\Rightarrow$ < substituting `this` for *rcvr* and `p` for *e* in the last two conjuncts >

rcvr $\equiv$ `this` $\wedge$ *e* $\equiv$ `p` $\wedge$ S<*e* != `null`>

$\wedge$ *isPivot*(`this.rPvt`, *typeOf*(`this`)) $\wedge$ *isPivot*(`this.pPvt`, *typeOf*(`p`))

$\Rightarrow$ < by the induction hypothesis >

rcvr $\equiv$ `this` $\wedge$ *e* $\equiv$ `p`

$\wedge$ S<`this` != `p`> $\wedge$ (!*isPivot*(`this.rPvt`, *typeOf*(`this`)) $\vee$ S<`p` != `this.rPvt`>)

$\wedge$ (!*isPivot*(`this.pPvt`, *typeOf*(`p`)) $\vee$ S<`this` != `p.pPvt`>)))

$\Rightarrow$ < substituting *rcvr* for `this` and *e* for `p` >

S<*rcvr* != *e*> $\wedge$ (!*isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) $\vee$ S<*e* != *rcvr*`.rPvt`>)

$\wedge$ (!*isPivot*(`this.pPvt`, *typeOf*(*e*)) $\vee$ S<*rcvr* != *e*`.pPvt`>)

Case 14: *rcvr* $\equiv$ `p` $\wedge$ *e* $\equiv$ `this`

If `p` $\in$ O, then this case holds from Case 2 above. Therefore, we assume that `p` $\notin$ O and there have been no assignments to `p`.

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`> $\wedge$ *assigns*(T, *m*) $\neq$ { }

$\wedge$!(*rcvr* $\equiv$ *e*) $\wedge$ (!*isPivot*(*e*, U) $\vee$!(*rcvr* $\equiv$ `this`)) $\wedge$ (!*isPivot*(*rcvr*, U) $\vee$!(*e* $\equiv$ `this`))

$\wedge$ *rcvr* $\equiv$ `p` $\wedge$ *e* $\equiv$ `this`

$\Rightarrow$ < the proof is analogous to Case 13 >

$S\langle rcvr \mathrel{!}= e\rangle \wedge (S\langle e == \texttt{null}\rangle \vee (S\langle e \mathrel{!}= rcvr\texttt{.}rPvt\rangle \wedge S\langle rcvr \mathrel{!}= e\texttt{.}pPvt\rangle))$

Case 15: $isPivot(rcvr, U) \wedge e \equiv \texttt{p}$

If $\texttt{p} \in O$, then this case holds from Case 8 above. Therefore, we assume that $\texttt{p} \notin O$ and there have been no assignments to `p`. Without loss of generality, we also assume that $rcvr \equiv \texttt{this.}f$ since it is a pivot field of the receiver.

$typeOf(e) \in TypeId \wedge S\langle rcvr \mathrel{!}= \texttt{null}\rangle \wedge S\langle e \mathrel{!}= \texttt{null}\rangle \wedge assigns(T, m) \neq \{\}$
$\wedge\ !(rcvr \equiv e) \wedge (!isPivot(e, U) \vee !(rcvr \equiv \texttt{this})) \wedge (!isPivot(rcvr, U) \vee !(e \equiv \texttt{this}))$
$\wedge\ isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f$

$\Rightarrow <$ logic $>$
$isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f \wedge S\langle e \mathrel{!}= \texttt{null}\rangle$

$\Rightarrow <$ by the induction hypothesis $>$
$isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f \wedge S\langle e \mathrel{!}= \texttt{null}\rangle$
$\wedge\ S\langle \texttt{this} \mathrel{!}= \texttt{p}\rangle$

$\Rightarrow <$ by the induction hypothesis and since $\texttt{this.}f$ is a pivot field of the receiver $>$
$isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f \wedge S\langle e \mathrel{!}= \texttt{null}\rangle$
$\wedge\ S\langle \texttt{this} \mathrel{!}= \texttt{p}\rangle \wedge S\langle \texttt{p} \mathrel{!}= \texttt{this.}f\rangle$

$\Rightarrow <$ substituting e for `p` $>$
$isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f \wedge S\langle e \mathrel{!}= \texttt{null}\rangle$
$\wedge\ S\langle \texttt{this} \mathrel{!}= e\rangle \wedge S\langle e \mathrel{!}= \texttt{this.}f\rangle$

$\Rightarrow <$ if $isPivot(\texttt{this.}pPvt, typeOf(e))$, then $S\langle \texttt{this.}f \mathrel{!}= e\texttt{.}pPvt\rangle$ since
$S\langle \texttt{this} \mathrel{!}= e\rangle$, i.e., they are pivot fields of different objects so $\texttt{this.}f$ and $e\texttt{.}pPvt$ cannot reference the same object $>$
$isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f \wedge S\langle e \mathrel{!}= \texttt{null}\rangle \wedge S\langle \texttt{this} \mathrel{!}= e\rangle$
$\wedge\ S\langle e \mathrel{!}= \texttt{this.}f\rangle \wedge (!isPivot(\texttt{this.}pPvt, typeOf(e)) \vee S\langle \texttt{this.}f \mathrel{!}= e\texttt{.}pPvt\rangle)$

$\Rightarrow <$ substituting $rcvr$ for $\texttt{this.}f$ in the last two conjuncts $>$
$isPivot(rcvr, U) \wedge e \equiv \texttt{p} \wedge rcvr \equiv \texttt{this.}f \wedge S\langle e \mathrel{!}= \texttt{null}\rangle \wedge S\langle \texttt{this} \mathrel{!}= e\rangle$
$\wedge\ S\langle e \mathrel{!}= rcvr\rangle \wedge (!isPivot(\texttt{this.}pPvt, typeOf(e)) \vee S\langle rcvr \mathrel{!}= e\texttt{.}pPvt\rangle)$

$\Rightarrow <$ if $isPivot(\texttt{this.}rPvt, typeOf(\texttt{this.}f))$, then $S\langle \texttt{p} \mathrel{!}= \texttt{this.}f\texttt{.}rPvt\rangle$ since
otherwise, in some previous call in the chain, $\texttt{this.}f$ would have to have been the receiver in order to access the pivot field $rPvt$ and pass $\texttt{this.}f\texttt{.}rPvt$ as an argument; also, the current receiver `this` would have to be the other argument in that same previous call in order to have the same object as the receiver again in $U\texttt{.}n$ (by the Self Callback Lemma 5.21);
however, the induction hypothesis implies (by the Self-Call Pivot Lemma 5.22 and the This-Argument Call Lemma 5.23) that this combination of argument objects is not allowed in any of the previous calls in the chain when there are side-effects to either the receiver or formal parameter. $>$

isPivot(*rcvr*, U) ∧ *e* ≡ p ∧ *rcvr* ≡ this.*f* ∧ S<*e* != null> ∧ S<this != *e*>
∧ S<*e* != *rcvr*> ∧ (!*isPivot*(this.*pPvt*, *typeOf*(*e*)) ∨ S<*rcvr* != *e*.*pPvt*>)
∧ (!*isPivot*(this.*rPvt*, *typeOf*(this.*f*)) ∨ S<p != this.*f*.*rPvt*>)

⇒ < logic and substituting *rcvr* for this.*f* and *e* for p in the last conjunct >

S<*rcvr* != *e*> ∧ (!*isPivot*(this.*pPvt*, *typeOf*(*e*)) ∨ S<*rcvr* != *e*.*pPvt*>)
∧ (!*isPivot*(this.*rPvt*, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*.*rPvt*>)

Case 16: *rcvr* ≡ p ∧ *isPivot*(*e*, U)

typeOf(*e*) ∈ *TypeId* ∧ S<*rcvr* != null> ∧ S<*e* != null> ∧ *assigns*(T, *m*) ≠ { }
∧ !(*rcvr* ≡ *e*) ∧ (!*isPivot*(*e*, U) ∨ !(*rcvr* ≡ this)) ∧ (!*isPivot*(*rcvr*, U) ∨ !(*e* ≡ this))
∧ *rcvr* ≡ p ∧ *isPivot*(*e*, U)

⇒ < analogous to the proof of Case 15 >

S<*rcvr* != *e*> ∧ (!*isPivot*(this.*rPvt*, *typeOf*(*rcvr*)) ∨ S<*e* != *rcvr*.*rPvt*>)
∧ (!*isPivot*(this.*pPvt*, *typeOf*(*e*)) ∨ S<*rcvr* != *e*.*pPvt*>)

■

The Actual Parameter Aliasing Lemma 5.24 above proves that the receiver and formal parameter will not share state when either have fields that can be assigned to. The soundness of our technique requires that all side-effects to the receiver be specified separately from any side-effects to fields of a formal parameter in the assignable clause of self-calls and super-calls. The next lemma proves the property needed for the soundness of our technique, and in particular, the soundness of our use of the assignable clause. The next lemma shows that all allowed assignments to fields of the receiver are specified in the assignable clause, i.e., our technique does not allow assignments to fields of a formal parameter to modify the state of the receiver object in a self-call or super-call. Thus assignments to fields of a formal parameter object cannot modify the state of the receiver.

Lemma 5.25 (No Overlapping Assignable Fields): Let *rcvr*.*m*(*e*) be a method call allowed by our technique in some method U.*n*. Let S ∈ *State* be an intermediate state at the point of this call during the execution of U.*n*.

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ S<*rcvr* != null>

then *rcvr* and *e* do not share mutable state that is modifiable by *rcvr*.*m*(*e*).

Proof:

The lemma is clearly true if T.*m* does not have permission to assign to fields of either the receiver or the formal parameter, so assume *assigns*(T, *m*) ≠ { }; similarly, the lemma is also true when *typeOf*(*e*) ∉ *TypeId* and S<*e* == null>, so we will assume *typeOf*(*e*) ∈ *TypeId* and S<*e* != null>.

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ *assigns*(T, *m*) ≠ { } ∧ *typeOf*(*e*) ∈ *TypeId*
∧ S<*rcvr* != null> ∧ S<*e* != null>

⇒ < Let this.*rPvt* be a pivot field of *rcvr* and this.*pPvt* be a pivot field of *e* >

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ *assigns*(T, *m*) ≠ { } ∧ *typeOf*(*e*) ∈ *TypeId*
∧ S<*rcvr* != null> ∧ S<*e* != null>

$\wedge$ *isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) $\wedge$ *isPivot*(`this.pPvt`, *typeOf*(*e*))

$\Rightarrow$ < by the Actual Parameter Aliasing Lemma 5.24 and since *rPvt* and *pPvt* are pivot fields of *rcvr* and *e* respectively >

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`>

$\wedge$ S<*rcvr* != *e*> $\wedge$ (!*isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) $\vee$ S<*e* != *rcvr*`.rPvt`>)

$\wedge$ (!*isPivot*(`this.pPvt`, *typeOf*(*e*)) $\vee$ S<*rcvr* != *e*`.pPvt`>)

$\wedge$ *isPivot*(`this.rPvt`, *typeOf*(*rcvr*)) $\wedge$ *isPivot*(`this.pPvt`, *typeOf*(*e*))

$\Rightarrow$ < logic >

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`>

$\wedge$ S<*rcvr* != *e*> $\wedge$ S<*e* != *rcvr*`.rPvt`> $\wedge$ S<*rcvr* != *e*`.pPvt`>

$\Rightarrow$ < Let `this.`*f* be a pivot field of `this` and `this.`*g* be a pivot field of `p` >

typeOf(*e*) $\in$ *TypeId* $\wedge$ S<*rcvr* != `null`> $\wedge$ S<*e* != `null`>

$\wedge$ S<*rcvr* != *e*> $\wedge$ S<*e* != *rcvr*`.rPvt`> $\wedge$ S<*rcvr* != *e*`.pPvt`>

$\wedge$ *isPivot*(`this.`*f*, *typeOf*(`this`)) $\wedge$ *isPivot*(`this.`*g*, *typeOf*(`p`))

$\Rightarrow$ < by the Self-Call Pivot Lemma 5.22 and the This-Argument Call Lemma 5.23 >

S<*rcvr* != *e*> $\wedge$ S<*e* != *rcvr*`.rPvt`> $\wedge$ S<*rcvr* != *e*`.pPvt`>

$\wedge$! (S<*e* == `this`> $\wedge$ S<*rcvr* == `this.`*f*>)

$\wedge$! (S<*rcvr* == `this`> $\wedge$ S<*e* == `this.`*g*>)

We use this last assertion to show that the actual parameters do not share mutable state that is assignable during the call of *rcvr*`.`*m*(*e*).

During a method call, the state of the receiver can only be changed through direct assignments to fields of the receiver or by object-calls on pivot fields of the receiver (see the syntax of assignments and method calls in Figure 4.5 and our assumptions in subsection 1.6.6). Therefore, the only way changes to the state of object *e* could change the state of *rcvr* would be if *e* and *rcvr* are aliases or if *e* is an alias of a pivot object internal to *rcvr*. However, S<*rcvr* != *e*> and S<*e* != *rcvr*`.rPvt`> disallows this situation from occurring in a valid call chain. Similarly, S<*rcvr* != *e*> and S<*rcvr* != *e*`.pPvt`> prevents changes to the state of *rcvr* from changing the state of *e*.

We also need to be sure that the objects referenced by the receiver and formal parameter do not share mutable state that is internal to pivot objects of either argument, i.e., neither argument object can be a pivot of a pivot object of the other parameter, etc. This situation is not allowed by the last two conjuncts, e.g., ! (S<*e* == `this`> $\wedge$ S<*rcvr* == `this.pvt`>). That is, in our technique, only pivot fields of the receiver can be directly accessed by methods (see the syntax of variable references in Figure 4.5 and our assumptions in subsection 1.6.6). Therefore, a pivot object of the current receiver would have to be the receiver in an object-call in order for a method to access a pivot field of a pivot of the current receiver; also, the formal parameter would have to reference the current receiver (otherwise, by the Self Callback Lemma 5.21, no callback on the current receiver would be possible). But this combination of actual parameters is not allowed, by the last two conjuncts, unless there are no

side-effects in *rcvr*`.`*m*(*e*), i.e., unless there are no side-effects in a call such as `this.`*pvt*`.`*m*(`this`) or `this.`*m*(`this.`*pvt*). Therefore, in our technique, assignments that change the state of *e* in the call *rcvr*`.`*m*(*e*) cannot change the state of *rcvr* and assignments that change the state of *rcvr* cannot change the state of *e*. ■

The next two lemmas prove relationships between the functions *selfAssigns*, *parmAssigns* and *assigns* for use in subsequent lemmas and theorems.

Lemma 5.26 (Self Assignments): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T.

For all *f*, *g* ∈ *VarId*:

(1) `this.`*f* ∈$_a$ *selfAssigns*(`this`, T, *m*) ⇔ `this.`*f* ∈$_a$ *assigns*(T, *m*)

(2) `this.`*f*`.`*g* ∈$_a$ *selfAssigns*(`this`, T, *m*) ⇔ `this.`*f*`.`*g* ∈$_a$ *assigns*(T, *m*),

Proof:

Part 1:

`this.`*f* ∈$_a$ *selfAssigns*(`this`, T, *m*)

⇔ < definition of *selfAssigns* of Figure 5.4 >

`this.`*f* ∈$_a$ { `this.`*g* | `this.`*g* ∈$_a$ *assigns*(T, *m*) }[`this` ← `this`]

⇔ < definition of substitution >

`this.`*f* ∈$_a$ { `this.`*g* | `this.`*g* ∈$_a$ *assigns*(T, *m*) }

⇔ < definition of set membership and ∈$_a$ >

`this.`*f* ∈$_a$ *assigns*(T, *m*)

Part 2:

`this.`*f*`.`*g* ∈$_a$ *selfAssigns*(`this`, T, *m*)

⇔ < definition of *selfAssigns* of Figure 5.4 >

`this.`*f*`.`*g* ∈$_a$ { `this.`*g*`.`*x* | `this.`*g*`.`*x* ∈$_a$ *assigns*(T, *m*) }[`this` ← `this`]

⇔ < definition of substitution >

`this.`*f*`.`*g* ∈$_a$ { `this.`*g*`.`*x* | `this.`*g*`.`*x* ∈$_a$ *assigns*(T, *m*) }

⇔ < definition of set membership and ∈$_a$ >

`this.`*f*`.`*g* ∈$_a$ *assigns*(T, *m*)

■

Lemma 5.27 (Parameter Field Assignments): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T.

For all *f*, *g* ∈ *VarId*:

`p.`*f* ∈$_a$ *parmAssigns*(`p`, T, *m*) ⇔ `p.`*f* ∈$_a$ *assigns*(T, *m*)

Proof:

`p.`*f* ∈$_a$ *parmAssigns*(`p`, T, *m*)

⇔ < definition of *parmAssigns* of Figure 5.4 >

$\texttt{p}.f \in_a \{ \texttt{p}.g \mid \texttt{p}.g \in_a \mathit{assigns}(\mathrm{T}, m) \}[\texttt{p} \leftarrow \texttt{p}]$

$\Leftrightarrow <$ definition of substitution $>$

$\texttt{p}.f \in_a \{ \texttt{p}.g \mid \texttt{p}.g \in_a \mathit{assigns}(\mathrm{T}, m) \}$

$\Leftrightarrow <$ definition of set membership and $\in_a >$

$\texttt{p}.f \in_a \mathit{assigns}(\mathrm{T}, m)$

■

The next lemma proves that if a method $\mathrm{U}.n$ satisfies the *validInvariant* predicate, then so will any super-calls made from $\mathrm{U}.n$. This lemma is used to simplify the proof of Lemma 5.29 which is used in the proof of the Valid Invariant Theorem 5.30.

Lemma 5.28 (Super-Call Invariant): Let $\texttt{super}.m(e)$ be a super-call allowed by our technique in some method $\mathrm{U}.n$. Let T1, T $\in$ *TypeId* be valid classes allowed by the rules of our technique.

If $\mathrm{T} = \mathit{whereMethodDecl}(\mathit{superOf}(\mathrm{U}), m) \wedge \mathrm{T1} < \mathrm{T} \wedge \mathit{validInvariant}(\mathrm{T1}, \mathrm{U}, n)$,

then $\mathit{validInvariant}(\mathrm{T1}, \mathrm{T}, m)$.

Proof:

$\mathrm{T1} < \mathrm{T} \wedge \mathit{validInvariant}(\mathrm{T1}, \mathrm{U}, n)$

$\Rightarrow <$ definition of *validInvariant* of Figure 5.5 $>$

$(\forall\ \texttt{this}.f \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f \notin_a \mathit{assigns}(\mathrm{U}, n))$

$\wedge\ (\forall\ \texttt{this}.f.g \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f.g \notin_a \mathit{assigns}(\mathrm{U}, n))$

$\Rightarrow <$ from the T-SupCall rule of Figure 5.1 and since the call is made from $\mathrm{U}.n >$

$\mathit{selfAssigns}(\texttt{this}, \mathrm{T}, m) \subseteq_a \mathit{assigns}(\mathrm{U}, n)$

$\wedge\ (\forall\ \texttt{this}.f \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f \notin_a \mathit{assigns}(\mathrm{U}, n))$

$\wedge\ (\forall\ \texttt{this}.f.g \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f.g \notin_a \mathit{assigns}(\mathrm{U}, n))$

$\Rightarrow <$ from the definition of $\in_a$ and $\subseteq_a >$

$(\forall\ \texttt{this}.f \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f \notin_a \mathit{selfAssigns}(\texttt{this}, \mathrm{T}, m))$

$\wedge\ (\forall\ \texttt{this}.f.g \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f.g \notin_a \mathit{selfAssigns}(\texttt{this}, \mathrm{T}, m))$

$\Rightarrow <$ by the Self Assignments Lemma 5.26 $>$

$(\forall\ \texttt{this}.f \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f \notin_a \mathit{assigns}(\mathrm{T}, m))$

$\wedge\ (\forall\ \texttt{this}.f.g \in \mathit{accessed}(\mathit{invOf}(\texttt{TEnv}(\mathrm{T1})) : \texttt{this}.f.g \notin_a \mathit{assigns}(\mathrm{T}, m))$

$\Rightarrow <$ definition of *validInvariant* in Figure 5.5 $>$

$\mathit{validInvariant}(\mathrm{T1}, \mathrm{T}, m)$

■

Our next lemma proves that if our technique allows a superclass method to be called, then that method does not at any time during execution invalidate any of the subclass portions of the run-time type invariant.

Lemma 5.29 (Valid Subclass Invariant): Let $\mathit{rcvr}.m(e)$ be a method call allowed by our technique in some method $\mathrm{U}.n$. Let T1, T $\in$ *TypeId* be valid classes allowed by the rules of our technique.

If *rcvr*.*m*(*e*) invokes method T.*m* ∧ T1 < T,

then *validInvariant*(T1, T, *m*).

Proof:

There are two cases, either method T.*m* is overridden or T.*m* is not overridden in any subclass of T.

Case 1: method *m* is not overridden in any subclass of T

This case handles self-calls, super-calls, and object-calls when method *m* is not overridden.

rcvr.*m*(*e*) invokes method T.*m* ∧ T1 < T ∧ ! *isOverridden*(T1, *m*)

⇒ < since T.*m* is not overridden in T1 or in any superclass of T1 and by the definition of *allMethodsIn* of Figure 5.6 >

T1 < T ∧ ! *isOverridden*(T1, *m*) ∧ T.*m* ∈ *allMethodsIn*(*superOf*(T1))

⇒ < by the Method Overriding Lemma 5.5 >

noAddSideEffects(T1, T, *m*) ∧ *validInvariant*(T1, T, *m*)

⇒ < logic >

validInvariant(T1, T, *m*)

Case 2: T.*m* is overridden in some subclass of T

Since T.*m* is overridden, it can only be super-called; so the proof will be by induction on the number of calls in a valid object-call segment that precedes the super-call of T.*m*.

Basis: *k* = 1, i.e., <U.*n*, T**::***m*> is a valid object-call segment.

We split the proof into three cases depending on where T1 is relative to U.

Basis Case 1: U ≡ T1 < T

`super`.*m*(*e*) invokes method T.*m* ∧ T1 ≡ U < T

⇒ < the super-call of T**::***m* must satisfy the *okToSuperCall*(U, T, *m*) predicate since that predicate is an antecedent in the T-SupCall rule of Figure 5.1 and because the super-call was made from method U.*n* >

okToSuperCall(U, T, *m*) ∧ T1 ≡ U

⇒ < substituting T1 for U >

okToSuperCall(T1, T, *m*)

⇒ < definition of *okToSuperCall* in Figure 5.4 >

validInvariant(T1, T, *m*) ∧ *validCalls*(T1, T, *m*)

⇒ < logic >

validInvariant(T1, T, *m*)

Basis Case 2: U < T1 < T

`super`.*m*(*e*) invokes method T.*m* ∧ U < T1 < T

⇒ < the ability to directly super-call T.*m* from U.*n*, when U < T, means that T.*m* was not overridden in any classes in the hierarchy between U and T, e.g., in T1 >

U < T1 < T ∧ ! *isOverridden*(T1, *m*)

$\Rightarrow <$ since T.*m* is not overridden in any classes between U and T and
by the definition of *allMethodsIn* of Figure 5.6 $>$

U < T1 < T $\wedge$! *isOverridden*(T1, *m*) $\wedge$ T.*m* $\in$ *allMethodsIn*(*superOf*(T1))

$\Rightarrow <$ from the Method Overriding Lemma 5.5 $>$

noAddSideEffects(T1, T, *m*) $\wedge$ *validInvariant*(T1, T, *m*)

$\Rightarrow <$ logic $>$

validInvariant(T1, T, *m*)

Basis Case 3: T1 < U < T

`super`.*m*(*e*) invokes method T.*m* $\wedge$ T1 < U < T

$\Rightarrow <$ The ability to directly object-call U.*n*, means that U.*n* was not overridden in any subclass of U $>$

T1 < U < T $\wedge$! *isOverridden*(T1, *n*)

$\Rightarrow <$ since U.*n* is not overridden in any subclass of U and by the definition of *allMethodsIn* of Figure 5.6 $>$

T1 < U < T $\wedge$! *isOverridden*(T1, *n*) $\wedge$ U.*n* $\in$ *allMethodsIn*(*superOf*(T1))

$\Rightarrow <$ from the Method Overriding Lemma 5.5 $>$

T1 < U < T $\wedge$ *noAddSideEffects*(T1, U, *n*) $\wedge$ *validInvariant*(T1, U, *n*)

$\Rightarrow <$ logic $>$

T1 < U < T $\wedge$ *validInvariant*(T1, U, *n*)

$\Rightarrow <$ since, from the premise of the lemma, `super`.*m*(*e*) invokes T**::***m* from U.*n* $>$

T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ T1 < U < T $\wedge$ *validInvariant*(T1, U, *n*)

$\Rightarrow <$ by the Super-Call Invariant Lemma 5.28 $>$

validInvariant(T1, T, *m*)

Induction Step: Let *k* = $\mathbb{N}$.

The induction hypothesis asserts that

If `super`.*m*(*e*) invokes method T.*m* $\wedge$ T1 < T
$\wedge$ *k* < $\mathbb{N}$ $\wedge$ <U1.*n1*, U2.*n2*, ... , U.*n*, T**::***m*> is a valid object-call segment
with U.*n* the *k*th call in the chain,

then *validInvariant*(T1, T, *m*).

There are two cases, either U.*n* is a self-call or a super-call.

Induction Case 1: U.*n* is a self-call

If U.*n* is a self-call, then U.*n* can be object-called. Thus <U.*n*, T**::***m*> is a valid object-call segment. Hence, the proof is the same as given above for the Basis.

Induction Case 2: U.*n* is a super-call

Since U.*n* is a super-call, we start calculating from the induction hypothesis.

`super`.*m*(*e*) invokes method T.*m* from U.*n* $\wedge$ T1 < T $\wedge$ *validInvariant*(T1, U, *n*)

$\Rightarrow <$ since `super`.*m*(*e*) invokes T.*m* from U.*n* $>$

T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ T1 < T $\wedge$ *validInvariant*(T1, U, *n*)

$\Rightarrow$ < by the Super-Call Invariant Lemma 5.28 >

validInvariant(T1, T, *m*)

■

We now prove the main theorem of this subsection; this theorem proves that the superclass methods (that our technique allows to be executed) do not at any time during execution invalidate any of the subclass portions of the run-time type invariant. Therefore, when the invariant of the static type of the receiver is established during a super-call, the invariant of the run-time type of the receiver is also established.

Theorem 5.30 (Valid Invariant): Let U.*n* be the method being executed. Let D be the run-time type of the current receiver. Let $S \in$ *State* be the pre-state prior to the execution of U.*n* and let $S' \in$ *State* be an intermediate state during the execution of U.*n*.

If $D \leq U \wedge S<inv(D)> \wedge S'<inv(U)>$,

then $S'<inv(D)>$.

Proof:

From the definition of *inv* in Figure 4.11, we know that

$(\forall$ T1, D $\in$ *TypeId* : $D \leq T1 \wedge S<inv(D)> \Rightarrow S<invOf(T1)>)$.

Thus the invariant of class D is only invalidated if there is a superclass of D with an invariant that does not hold.

From the Invariant Clause Lemma 5.1, we have that

$S'<inv(U)> \Rightarrow S'<invOf(T1)>$ for all T1 such that $D \leq U \leq T1$.

Therefore, we still need to show that

$S'<invOf(T1)>$ = true for all T1 such that $D \leq T1 < U$.

$D \leq T1 < U \wedge S<inv(D)> \wedge S'<inv(U)>$

$\Rightarrow$ < the execution of U.*n* has been allowed by our technique, so the conclusion of the Valid Subclass Invariant Lemma 5.29 must hold for U.*n* >

validInvariant(T1, U, *n*) $\wedge$ $D \leq T1 < U \wedge S<inv(D)> \wedge S'<inv(U)>$

$\Rightarrow$ < definition of *validInvariant* in Figure 5.5 >

$(\forall$ `this`.*f* $\in$ *accessed*(*invOf*(`TEnv`(T1)) : `this`.*f* $\notin_a$ *assigns*(U, *n*))

$\wedge$ $(\forall$ `this`.*f*.*g* $\in$ *accessed*(*invOf*(`TEnv`(T1)) : `this`.*f*.*g* $\notin_a$ *assigns*(U, *n*))

$\wedge$ $D \leq T1 < U \wedge S<inv(D)> \wedge S'<inv(U)>$

$\Rightarrow$ < the fields assignable during U.*n* are not accessed by T1's part of the invariant and by the No Overlapping Assignable Fields Lemma 5.25, any side-effects to fields of the object referenced by the formal parameter do not change the state of the receiver >

$S<invOf(T1)> = S'<invOf(T1)> \wedge D \leq T1 < U \wedge S<inv(D)>$

$\Rightarrow$ < since $S<inv(D) \Rightarrow S<inv(T1)>$ by the Invariant Clause Lemma 5.1 >

S<*invOf*(T1)> = S′<*invOf*(T1)> ∧ D ≤ T1 < U ∧ S<*inv*(T1)>

⇒ < since S<*inv*(T1)> ⇒ S<*invOf*(T1)> from the definition of *inv* in Figure 4.11 >

S<*invOf*(T1)> = S′<*invOf*(T1)> ∧ D ≤ T1 < U ∧ S<*invOf*(T1)>

⇒ < since S<*invOf*(T1)> and S<*invOf*(T1)> = S′<*invOf*(T1)> hold >

S′<*invOf*(T1)>

■

5.2.4 The Soundness of Our Alias Control Technique

In this subsection, we first prove the Owner Aliasing Theorem that shows that our alias control technique prevents any pair of owner variables visible in the same context from being aliases of the same object. We then prove that our technique does not allow unexpected side-effects, i.e., the only fields that change during execution of a method are those permitted by its assignable clause.

5.2.4.1 The Owner Aliasing Theorem

Lemma 5.31 (Self Owner Aliasing): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in T.*m*. Let *vr* be a variable accessible at that same point in T.*m*. Let S ∈ *State* be an intermediate state also at the same point during the execution of T.*m*.

If !(`this` ≡ *vr*) ∧ *typeOf*(*vr*) ∈ *TypeId* ∧ *isOwner*(*vr*, T, O) ∧ *assigns*(T, *m*) ≠ { },

then S<`this` != *vr*>.

Proof:

!(`this` ≡ *vr*) ∧ *typeOf*(*vr*) ∈ *TypeId* ∧ *isOwner*(*vr*, T, O) ∧ *assigns*(T, *m*) ≠ { }

⇒ < definition of *isOwner* in Figure 5.5 >

!(`this` ≡ *vr*) ∧ (*vr* ∈ O ∨ *vr* ≡ `this` ∨ *vr* ≡ `p` ∨ *isPivot*(*vr*, T))

∧ *typeOf*(*vr*) ∧ *assigns*(T, *m*) ≠ { }

⇒ < logic >

!(`this` ≡ *vr*) ∧ (*vr* ∈ O ∨ *vr* ≡ `p` ∨ *isPivot*(*vr*, T))

∧ *typeOf*(*vr*) ∧ *assigns*(T, *m*) ≠ { }

⇒ < by the Self New Object Aliasing Lemma 5.14 (when *vr* ∈ O),
by the Actual Parameter Aliasing Lemma 5.24 (when *vr* ≡ `p`), and
by the Self Pivot Aliasing Lemma 5.11 (when *isPivot*(*vr*, T)) >

S<`this` != *vr*>

■

Lemma 5.32 (Parameter Owner Aliasing): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in T.*m*. Let *vr* be a variable accessible at that same point in T.*m*. Let S ∈ *State* be an intermediate state also at the same point during the execution of T.*m*.

If !(p ≡ *vr*) ∧ *typeOf*(p) ∈ *TypeId* ∧ *typeOf*(*vr*) ∈ *TypeId* ∧ *isOwner*(*vr*, T, O)
∧ *assigns*(T, *m*) ≠ { } ∧ (S<p != null> ∨ S<*vr* != null>),
then S<p != *vr*>.
Proof:
!(p ≡ *vr*) ∧ *typeOf*(p) ∈ *TypeId* ∧ *typeOf*(*vr*) ∈ *TypeId* ∧ *isOwner*(*vr*, T, O)
∧ *assigns*(T, *m*) ≠ { } ∧ (S<p != null> ∨ S<*vr* != null>),
⇒< definition of *isOwner* in Figure 5.5 >
!(p ≡ *vr*) ∧ (*vr* ∈ O ∨ *vr* ≡ this ∨ *vr* ≡ p ∨ *isPivot*(*vr*, T))
∧ *typeOf*(p) ∈ *TypeId* ∧ *typeOf*(*vr*) ∈ *TypeId*
∧ *assigns*(T, *m*) ≠ { } ∧ (S<p != null> ∨ S<*vr* != null>),
⇒< logic >
!(p ≡ *vr*) ∧ (*vr* ∈ O ∨ *vr* ≡ this ∨ *isPivot*(*vr*, T))
!∧ *typeOf*(p) ∈ *TypeId* ∧ *typeOf*(*vr*) ∈ *TypeId*
∧ *assigns*(T, *m*) ≠ { } ∧ (S<p != null> ∨ S<*vr* != null>),
⇒< by the Parameter New Object Aliasing Lemma 5.15 (when *vr* ∈ O) and
by the Actual Parameter Aliasing Lemma 5.24 (when *vr* ≡ this or *isPivot*(*vr*, T)) >
S<p != *vr*>
■

Lemma 5.33 (Pivot Owner Aliasing): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in T.*m*. Let *vr1* and *vr2* be variables accessible at that same point in T.*m*. Let S ∈ *State* be an intermediate state also at the same point during the execution of T.*m*.

If !(*vr1* ≡ *vr2*) ∧ *isPivot*(*vr1*, T) ∧ *typeOf*(*vr2*) ∈ *TypeId* ∧ *isOwner*(*vr2*, T, O)
∧ *assigns*(T, *m*) ≠ { } ∧ (S<*vr1* != null> ∨ S<*vr2* != null>),
then S<*vr1* != *vr2*>.
Proof:
!(*vr1* ≡ *vr2*) ∧ *isPivot*(*vr1*, T) ∧ *typeOf*(*vr2*) ∈ *TypeId* ∧ *isOwner*(*vr2*, T, O)
∧ *assigns*(T, *m*) ≠ { } ∧ (S<*vr1* != null> ∨ S<*vr2* != null>),
⇒< definition of *isOwner* in Figure 5.5 >
!(*vr1* ≡ *vr2*) ∧ *isPivot*(*vr1*, T) ∧ (*vr2* ∈ O ∨ *vr2* ≡ this ∨ *vr2* ≡ p ∨ *isPivot*(*vr2*, T))
∧ *typeOf*(*vr2*) ∈ *TypeId* ∧ *assigns*(T, *m*) ≠ { } ∧ (S<*vr1* != null> ∨ S<*vr2* != null>)
⇒< by the Owner Variable Aliasing Lemma 5.12 (when *vr2* ∈ O),
by the Actual Parameter Aliasing Lemma 5.24 (when *vr2* ≡ this or *vr2* ≡ p), and
by the Pivot Fields Aliasing Lemma 5.13 (when *isPivot*(*vr2*, T)) >
S<*vr1* != *vr2*>
■

Lemma 5.34 (New Owner Aliasing): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in T.*m*. Let *vr1* and *vr2* be variables accessible at that same point in T.*m*. Let S ∈ *State* be an intermediate state also at the same point during the execution of T.*m*.

If !(*vr1* ≡ *vr2*) ∧ *vr1* ∈ O ∧ *typeOf*(*vr2*) ∈ *TypeId* ∧ *isOwner*(*vr2*, T, O),
then S<*vr1* != *vr2*>.
Proof:

!(*vr1* ≡ *vr2*) ∧ *vr1* ∈ O ∧ *typeOf*(*vr2*) ∈ *TypeId* ∧ *isOwner*(*vr2*, T, O)
⇒ < definition of *isOwner* in Figure 5.5 >
!(*vr1* ≡ *vr2*) ∧ *vr1* ∈ O ∧ *typeOf*(*vr2*) ∈ *TypeId*
∧ (*vr2* ∈ O ∨ *vr2* ≡ `this` ∨ *vr2* ≡ `p` ∨ *isPivot*(*vr2*, T))
⇒ < by the Owner Variable Aliasing Lemma 5.12 (when *vr2* ∈ O or *isPivot*(*vr2*, T)),
by the Self New Object Aliasing Lemma 5.14 (when *vr2* ≡ `this`), and
by the Parameter New Object Aliasing Lemma 5.15 (when *vr2* ≡ `p`) >
S<*vr1* != *vr2*>

■

Theorem 5.35 (Owner Aliasing): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T. Let O be the set of owner variables from the T-rules of Figures 5.1 - 5.3 at an arbitrary point in T.*m*. Let *vr1* and *vr2* be variables accessible at that same point in T.*m*. Let S ∈ *State* be an intermediate state also at the same point during the execution of T.*m*.

If !(*vr1* ≡ *vr2*) ∧ *typeOf*(*vr1*) ∈ *TypeId* ∧ *isOwner*(*vr1*, T, O)
∧ *typeOf*(*vr2*) ∈ *TypeId* ∧ *isOwner*(*vr2*, T, O)
∧ *assigns*(T, *m*) ≠ { } ∧ (S<*vr1* != `null`> ∨ S<*vr2* != `null`>),
then S<*vr1* != *vr2*>.
Proof:

!(*vr1* ≡ *vr2*) ∧ *typeOf*(*vr1*) ∈ *TypeId* ∧ *typeOf*(*vr2*) ∈ *TypeId*
∧ *isOwner*(*vr1*, T, O) ∧ *isOwner*(*vr2*, T, O)
∧ *assigns*(T, *m*) ≠ { } ∧ (S<*vr1* != `null`> ∨ S<*vr2* != `null`>)
⇒ < definition of *isOwner* in Figure 5.5 >
!(*vr1* ≡ *vr2*) ∧ *typeOf*(*vr1*) ∈ *TypeId* ∧ *typeOf*(*vr2*) ∈ *TypeId*
∧ (*vr1* ∈ O ∨ *vr1* ≡ `this` ∨ *vr1* ≡ `p` ∨ *isPivot*(*vr1*, T))
∧ (*vr2* ∈ O ∨ *vr2* ≡ `this` ∨ *vr2* ≡ `p` ∨ *isPivot*(*vr2*, T))
∧ *assigns*(T, *m*) ≠ { } ∧ (S<*vr1* != `null`> ∨ S<*vr2* != `null`>)
⇒ < by the New Owner Aliasing Lemma 5.34 (when *vr1* ∈ O),
by the Self Owner Aliasing Lemma 5.31 (when *vr1* ≡ `this`),

by the Parameter Owner Aliasing Lemma 5.32 (when *vr1* ≡ `p`),
and by the Pivot Owner Aliasing Lemma 5.33 (when *isPivot*(*vr1*, T)) >
S<*vr1* != *vr2*>

■

The above theorem shows that if a method has side-effects, then no pair of owner variables, visible in the same context, can be aliases for the same object. Since side-effects can only be initiated through an object-call with an owner variable as the receiver, this theorem also means that, when the classes and methods satisfy our rules, verifiers can reason locally about aliasing and side-effects using the owner variable names.

5.2.4.2 The Assignable Clause Theorem

The first two lemmas in this section prove that calls at the end of a valid call chain have to satisify the assignable clause of the first call in the chain.

Lemma 5.36 (Self Assignable Chain): Let T ∈ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T.

If <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*> is a valid object-call segment for 0 ≤ *k*,
then *selfAssigns*(`this`, Uk, *nk*) ⊆$_a$ *assigns*(T, *m*)
Proof:

The proof will be by induction on the number of self-calls and super-calls in a valid object-call segment following the object-call of T . *m*.

Basis: *k* = 0, i.e., T . *m* makes no super-calls or self-calls.

<T . *m*> is a valid object-call segment
⇒ < by the Self Assignments Lemma 5.26 and the definition of ⊆$_a$ in Figure 5.6 >
selfAssigns(`this`, T, *m*) ⊆$_a$ *assigns*(T, *m*)

Induction Step:

The induction hypothesis asserts that

If *k* < N ∧ <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*> is a valid object-call segment,
then *selfAssigns*(`this`, Uk, *nk*) ⊆$_a$ *assigns*(T, *m*).

We now prove that the conclusion of the lemma holds when there are N self-calls or super-calls following the object-call of T . *m*.

k < N ∧ <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*, U . *n*> is a valid object-call segment
⇒ < by the induction hypothesis >
k < N ∧ <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*, U . *n*> is a valid object-call segment
∧ *selfAssigns*(`this`, Uk, *nk*) ⊆$_a$ *assigns*(T, *m*)
⇒ < by the T-Call and T-SupCall rule of Figure 5.1 since U . *n* is a super-call or self-call
from Uk . *nk* >
selfAssigns(`this`, U, *n*) ⊆$_a$ *assigns*(Uk, *nk*)

$\wedge$ *selfAssigns*(this, Uk, *nk*) $\subseteq_a$ *assigns*(T, *m*)

$\Rightarrow$ < by the Self Assignments Lemma 5.26 >

selfAssigns(this, U, *n*) $\subseteq_a$ *selfAssigns*(this, Uk, *nk*)

$\wedge$ *selfAssigns*(this, Uk, *nk*) $\subseteq_a$ *assigns*(T, *m*)

$\Rightarrow$ < from the definition in Figure 5.6, $\subseteq_a$ is transitive >

selfAssigns(this, U, *n*) $\subseteq_a$ *assigns*(T, *m*)

■

Lemma 5.37 (Parameter Assignable Chain): Let T $\in$ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T.

If <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*> is a valid call chain with $0 \leq k$

$\wedge$ the formal parameter references the same object in each call,

then *parmAssigns*(p, Uk, *nk*) $\subseteq_a$ *parmAssigns*(p, T, *m*)

Proof:

The proof will be by induction on the number of calls in a valid call chain following the call of T . *m*.

Basis: *k* = 0, i.e., T . *m* makes no super-calls or self-calls.

<T . *m*> is a valid object-call segment

$\Rightarrow$ < by the definition of $\subseteq_a$ in Figure 5.6 >

parmAssigns(p, T, *m*) $\subseteq_a$ *parmAssigns*(p, T, *m*)

Induction Step:

The induction hypothesis asserts that

If *k* < N $\wedge$ <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*> is a valid call chain

$\wedge$ the formal parameter references the same object in each call,

then *parmAssigns*(p, Uk, *nk*) $\subseteq_a$ *parmAssigns*(p, T, *m*).

We now prove that the conclusion of the lemma holds when there are N calls following the call of T . *m*.

k < N $\wedge$ <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*, U . *n*> is a valid object-call segment

$\Rightarrow$ < by the induction hypothesis >

k < N $\wedge$ <T . *m*, U1 . *n1*, U2 . *n2*, ... , Uk . *nk*, U . *n*> is a valid object-call segment

$\wedge$ *parmAssigns*(p, Uk, *nk*) $\subseteq_a$ *parmAssigns*(p, T, *m*)

$\Rightarrow$ < by the T-Call and T-SupCall rule of Figure 5.1 when U . *n* is called from Uk . *nk* >

parmAssigns(p, U, *n*) $\subseteq_a$ *assigns*(Uk, *nk*)

$\wedge$ *parmAssigns*(p, Uk, *nk*) $\subseteq_a$ *parmAssigns*(p, T, *m*)

$\Rightarrow$ < by the Parameter Field Assignments Lemma 5.27 >

parmAssigns(p, U, *n*) $\subseteq_a$ *parmAssigns*(p, Uk, *nk*)

$\wedge$ *parmAssigns*(p, Uk, *nk*) $\subseteq_a$ *parmAssigns*(p, T, *m*)

$\Rightarrow$ < $\subseteq_a$ is transitive from its definition in Figure 5.6 and because p references the same object >

parmAssigns(p, U, *n*) $\subseteq_a$ *parmAssigns*(p, T, *m*)

■

The lemmas in the rest of this subsection prove that our type checking rules enforce the assignable clause of the method specification; they culminate in the proof of the Assignable Clause Theorem 5.43.

Lemma 5.38: Let T $\in$ *TypeId* be a valid class allowed by the rules of our technique and let *m* be a method declared in T. Let S $\in$ *State* be the pre-state prior to the execution of T.*m* and let S′ $\in$ *State* be an intermediate state during the execution of T.*m*. Let D be the run-time type of the receiver of a call to T.*m*.

For all *f* $\in$ *allFieldsIn*(D):

If this.*f* $\notin_a$ *assigns*(T, *m*) $\wedge$ *f* is a concrete field

$\wedge$ T.*m* does not directly or indirectly make an object-call,

then S<this.*f*> = S′<this.*f*>.

Proof:

Since T.*m* does not directly or indirectly make an object-call, every valid call chain starting with T.*m* must be a valid object-call segment. Thus the proof will be by induction on the length of an arbitrary valid object-call segment starting with the object-call of T.*m*.

Basis: *k* = 0, i.e., T.*m* makes no calls.

When there are no self-calls or super-calls, only assignment statements in the body of method T.*m* can change the state of this.*f*, i.e., during the execution of T.*m*, there are no other places where assignments to fields of the receiver can occur because, in our technique, assignment to fields of objects other than the receiver are not allowed. However, an assignment to this.*f* requires that this.*f* $\in_a$ *assigns*(T, *m*) when the target is a field since this requirement is an antecedent in all of the rules for assignment given in Figure 5.2. Therefore, this.*f* cannot be the target of an assignment in T.*m* since this.*f* $\notin_a$ *assigns*(T, *m*). Hence, S<this.*f*> = S′<this.*f*>.

Induction Step: Let k = N.

The induction hypothesis asserts that

If this.*f* $\notin_a$ *assigns*(T, *m*) $\wedge$ T.*m* does not directly or indirectly make an object-call

$\wedge$ *f* is a concrete field

$\wedge$ *k* < N $\wedge$ <T.*m*, U1.*n1*, U2.*n2*, ... , Uk.*nk*> is a valid object-call segment,

then S<this.*f*> = S′<this.*f*>.

this.*f* $\notin_a$ *assigns*(T, *m*) $\wedge$ *f* is a concrete field

$\wedge$ *k* < N $\wedge$ <T.*m*, U1.*m1*, U2.*m2*, ... , Uk.*nk*, U.*n*> is a valid object-call segment

$\Rightarrow$ < by the Self Assignable Chain Lemma 5.36 >

selfAssigns(this, U, *n*) $\subseteq_a$ *assigns*(T, *m*) $\wedge$ this.*f* $\notin_a$ *assigns*(T, *m*) $\wedge$ *f* is a concrete field

$\wedge$ *k* < N $\wedge$ <T.*m*, U1.*m1*, U2.*m2*, ... , Uk.*nk*, U.*n*> is a valid object-call segment

$\Rightarrow$ < from the definition of $\in_a$ and $\subseteq_a$ >

`this`.*f* $\notin_a$ *selfAssigns*(`this`, U, *n*) $\wedge$ *f* is a concrete field

$\wedge$ *k* < N $\wedge$ <T.*m*, U1.*m1*, U2.*m2*, ... , Uk.*nk*, U.*n*> is a valid object-call segment,

$\Rightarrow$ < by the same reasoning as the basis, there are no assignments to `this`.*f* in U.*n* and by the induction hypothesis, none of the calls up through Uk.*nk* in the object-call segment can assign to `this`.*f* >

S<`this`.*f*> = S′<`this`.*f*>

■

Lemma 5.39 (Assignable Concrete Fields): Let D, T $\in$ *TypeId* be valid classes allowed by the rules of our technique and let *m* be a method declared in T. Let S $\in$ *State* be the pre-state prior to the execution of T.*m* and let S′ $\in$ *State* be an intermediate state during the execution of T.*m*. Let D be the run-time type of the receiver of a call to T.*m*.

For all *f* $\in$ *allFieldsIn*(D):

If `this`.*f* $\notin_a$ *assigns*(T, *m*) $\wedge$ *f* is a concrete field,

then S<`this`.*f*> = S′<`this`.*f*>.

Proof:

We know from Lemma 5.38 that the conclusion holds if T.*m* does not directly or indirectly make an object-call. Therefore, what is left to prove is that the conclusion holds when T.*m* makes object-calls. However, by the Self Callback Lemma 5.21, the only way an object-call can assign to fields of the current receiver is through a this-argument call, i.e., an object-call that passes `this` as the actual parameter corresponding to a formal parameter of the called method.

The proof will be by contradiction, i.e., we will prove that no valid call chain can violate this lemma. We can reason, without loss of generality, about call chains that start with the call of T.*m* and contain only one this-argument call since any valid call chain that could possibly violate this lemma must have such a suffix. Therefore, the call chain will start with a call of T.*m* followed by a this-argument call and end with a call that violates the lemma's conclusion. Also, because our technique does not allow assignment to fields of objects other than the receiver, this call chain must end with an object-call segment that assigns to `this`.*f*.

We now prove that the premise of the lemma `this`.*f* $\notin_a$ *assigns*(T, *m*) does not hold when the conclusion is false. We start calculating from the negation of the lemma's conclusion.

S<`this`.*f*> $\neq$ S′<`this`.*f*>

$\Rightarrow$ < since S<`this`.*f*> $\neq$ S′<`this`.*f*>, there must be a call, at the end of the chain, to a method U.*n* such that U.*n* has permission to assign to `this`.*f* >

S<`this`.*f*> $\neq$ S′<`this`.*f*> $\wedge$ `this`.*f* $\in_a$ *assigns*(U, *n*)

$\wedge$ <T.*m*, ... , U.*n*> is a valid call chain

$\Rightarrow$ < let T1.*m1* be a this-argument call from T.*m* >

S<`this`.*f*> $\neq$ S′<`this`.*f*> $\wedge$ `this`.*f* $\in_a$ *assigns*(U, *n*)

$\wedge$ <T . *m*, T1 . *m1*, ... , U . *n*> is a valid call chain

$\Rightarrow$ < since T1 . *m1* is a this-argument call from T . *m*, it must satisfy *parmAssigns* of the T-Call rule of Figure 5.1 for actual parameter `this` >

S<`this` . *f*> $\neq$ S′<`this` . *f*> $\wedge$ `this` . *f* $\in_a$ *assigns*(U, *n*)

$\wedge$ <T . *m*, T1 . *m1*, ... , U . *n*> is a valid call chain

$\wedge$ *parmAssigns*(`this`, T1, *m1*) $\subseteq_a$ *assigns*(T, *m*)

$\Rightarrow$ < some call in the chain between T1 . *m1* and U . *n* (say U1 . *n1*) has to make an object-call on its formal parameter since the call chain has to end with a valid object-call segment that assigns to `this` . *f* >

S<`this` . *f*> $\neq$ S′<`this` . *f*> $\wedge$ `this` . *f* $\in_a$ *assigns*(U, *n*) $\wedge$ U1 . *n1* is an object-call on `p`

$\wedge$ <T . *m*, T1 . *m1*, ... , Tk . *mk*, U1 . *n1*, ... , U . *n*> is a valid call chain

$\wedge$ *parmAssigns*(`this`, T1, *m1*) $\subseteq_a$ *assigns*(T, *m*)

$\Rightarrow$ < by Lemma 5.38, `this` . *f* $\in_a$ *assigns*(U1, *n1*) >

S<`this` . *f*> $\neq$ S′<`this` . *f*> $\wedge$ `this` . *f* $\in_a$ *assigns*(U1, *n1*) $\wedge$ U1 . *n1* is an object-call on `p`

$\wedge$ <T . *m*, T1 . *m1*, ... , Tk . *mk*, U1 . *n1*> is a valid call chain

$\wedge$ *parmAssigns*(`this`, T1, *m1*) $\subseteq_a$ *assigns*(T, *m*)

$\Rightarrow$ < since U1 . *n1* is an object-call on `p` from Tk . *mk*, it must satisfy *selfAssigns* of the T-Call rule of Figure 5.1 for receiver parameter `p` >

S<`this` . *f*> $\neq$ S′<`this` . *f*> $\wedge$ `this` . *f* $\in_a$ *assigns*(U1, *n1*) $\wedge$ U1 . *n1* is an object-call on `p`

$\wedge$ <T . *m*, T1 . *m1*, ... , Tk . *mk*, U1 . *n1*> is a valid call chain

$\wedge$ *parmAssigns*(`this`, T1, *m1*) $\subseteq_a$ *assigns*(T, *m*)

$\wedge$ *selfAssigns*(`p`, U1, *n1*) $\subseteq_a$ *assigns*(Tk, *mk*)

$\Rightarrow$ < by the Parameter Field Assignments Lemma 5.27 >

S<`this` . *f*> $\neq$ S′<`this` . *f*> $\wedge$ `this` . *f* $\in_a$ *assigns*(U1, *n1*) $\wedge$ U1 . *n1* is an object-call on `p`

$\wedge$ <T . *m*, T1 . *m1*, ... , Tk . *mk*, U1 . *n1*> is a valid call chain

$\wedge$ *parmAssigns*(`this`, T1, *m1*) $\subseteq_a$ *assigns*(T, *m*)

$\wedge$ *selfAssigns*(`p`, U1, *n1*) $\subseteq_a$ *parmAssigns*(`p`, Tk, *mk*)

$\Rightarrow$ < by the Parameter Assignable Chain Lemma 5.37 since each call between T1 . *m1* and Tk . *mk* must pass `p` as the formal parameter so `p` will be an alias of the receiver object from method T . *m* >

S<`this` . *f*> $\neq$ S′<`this` . *f*> $\wedge$ `this` . *f* $\in_a$ *assigns*(U1, *n1*) $\wedge$ U1 . *n1* is an object-call on `p`

$\wedge$ <T . *m*, T1 . *m1*, ... , Tk . *mk*, U1 . *n1*> is a valid call chain

$\wedge$ *parmAssigns*(`this`, T1, *m1*) $\subseteq_a$ *assigns*(T, *m*)

$\wedge$ *selfAssigns*(`p`, U1, *n1*) $\subseteq_a$ *parmAssigns*(`p`, Tk, *mk*)

$\wedge$ *parmAssigns*(`p`, Tk, *mk*) $\subseteq_a$ *parmAssigns*(`p`, T1, *m1*)

$\Rightarrow$ < substitute `this` for `p`, since all of the `p`'s in the different contexts are aliases of the receiver in T . *m* with respect to the program state during the call of U1 . *n1* >

S<this.f> ≠ S'<this.f> ∧ this.f ∈$_a$ *assigns*(U1, *n1*)
∧ <T.m, T1.*m1*, ... , Tk.*mk*, U1.*n1*> is a valid call chain
∧ *parmAssigns*(this, T1, *m1*) ⊆$_a$ *assigns*(T, *m*)
∧ *selfAssigns*(this, U1, *n1*) ⊆$_a$ *parmAssigns*(this, Tk, *mk*)
∧ *parmAssigns*(this, Tk, *mk*) ⊆$_a$ *parmAssigns*(this, T1, *m1*)
⇒ < from the definition (Figure 5.6), ⊆$_a$ is transitive >
S<this.f> ≠ S'<this.f> ∧ this.f ∈$_a$ *assigns*(U1, *n1*)
∧ *selfAssigns*(this, U1, *n1*) ⊆$_a$ *assigns*(T, *m*)
⇒ < by the Self Assignments Lemma 5.26 >
S<this.f> ≠ S'<this.f> ∧ this.f ∈$_a$ *selfAssigns*(this, U1, *n1*)
∧ *selfAssigns*(this, U1, *n1*) ⊆$_a$ *assigns*(T, *m*)
⇒ < from the definition of ∈$_a$ and ⊆$_a$ in Figure 5.6 >
S<this.f> ≠ S'<this.f> ∧ this.f ∈$_a$ *assigns*(T, *m*)
⇒ < the second conjunct contradicts the assumption that this.f ∉$_a$ *assigns*(T, *m*)
from the premise of the lemma >
false

So the conclusion of the lemma must hold.

■

Lemma 5.40 (Assignable Pivot Objects): Let D, T ∈ *TypeId* be valid classes allowed by the rules of our technique and let *m* be a method declared in T. Let S ∈ *State* be the pre-state prior to the execution of T.*m* and let S' ∈ *State* be an intermediate state during the execution of T.*m*. Let D be the run-time type of the receiver of a call to T.*m*.

For all f ∈ *allFieldsIn*(D) ∧ g ∈ *VarId*:
If D < T ∧ this.f.g ∉$_a$ *assigns*(T, *m*) ∧ f is a concrete field,
then S<this.f.g> = S'<this.f.g>.

Proof:

In our technique, to change the value of this.f.g, this.f must be a pivot field in the enclosing class. Our restrictions on assignment statements (Figure 5.2) require that a pivot field contain a reference to a newly created object or null so it is guaranteed to own the object it references. Also, only one variable in the program state can be the first to contain a reference to a newly created pivot object, so any other variables that reference a pivot object are read-only, i.e., cannot be used to initiate changes. That is, the pivot object referenced by this.f can only be changed through an object-call invoked on this.f or indirectly through an object-call if this.f is passed as an argument to a method with permission to modify p.g. Thus our technique allows a pivot object to be aliased, but only the owner of that pivot object can be used to initiate changes.

Our requirement that changes to the state of a pivot object only be done through its owner variable (pivot field) is analogous to our restriction on direct assignment to fields of the receiver, i.e., only

methods of the object containing a concrete field can directly assign to that field and, similarly, only methods of the object containing a pivot field can change the pivot object it references.

Therefore, this lemma can be proven using reasoning analogous to the proof of the Assignable Concrete Fields Lemma 5.39. That is, we can consider an object-call that is allowed to change `this.`f`.`g to be analogous to an assignment to `this.`f`.`g since such calls can only occur in methods of the object containing pivot field `this.`f. Note, however, that the proof has to also handle the situation where a pivot object is modified through a formal parameter of an object-call, but this part of the proof is analogous to the proof of the case of a this-argument call given in Lemma 5.39. Furthermore, we do not have to be concerned about a method initiating changes to one object and inadvertantly changing the state of a different object (by the Owner Aliasing Theorem 5.35 and by the No Overlapping Assignable Fields Lemma 5.25).

In our technique, g in the lemma must be a model field because concrete fields of objects other than the receiver are not accessible (see the Java-C syntax of variable references in Figure 4.5 and our assumptions in subsection 1.6.6). Furthermore, `this.`f`.`g must be a member of a public data group, say `this.`G, so `this.`f is a pivot field and so `this.`G can be listed in the assignable clause since concrete fields have protected visibility and cannot be mentioned in a public assignable clause (they are not be in scope). This restriction is further ensured by the JML-C syntax of Figure 4.6, i.e., the syntax does not allow `this.`f`.`g to be listed in an assignable clause.

Because g is a model field, our technique also has to ensure that this lemma holds for model fields. In particular, a model field must be allowed to change whenever any of the concrete fields that determine its value change; this requirement is formalized in the *validAssignable*, *validRepresents*, and *admissibleRep* predicates of Figure 5.7. Predicate *validAssignable* requires that whenever any concrete member of a data group is assignable, then that data group (i.e. model field) must also be assignable. To be sound, our technique also has to ensure that all concrete fields that determine the value of a model field g are members of g's data group; this requirement is formalized in the *validRepresents* predicate that requires that all model fields satisfy the *admissibleRep* predicate, i.e., contain all concrete fields accessed by the right hand side of the model field's represents clause (the details are used in the proof of the Assignable Model Fields Lemma 5.41).

Note also that we assume that the value of a model field is well-defined (see assumptions in subsection 1.6.6), i.e., the right hand side of a model field's represents clause must be an expression that, when evaluated, terminates and yields a value; also, this expression must not have side-effects. ■

Lemma 5.41 (Assignable Model Fields): Let D, T $\in$ *TypeId* be valid classes allowed by the rules of our technique and let m be a method declared in T. Let $\mathsf{S} \in$ *State* be the pre-state prior to the execution of T`.`m and let $\mathsf{S}' \in$ *State* be an intermediate state during the execution of T`.`m. Let D be the run-time type of the receiver of a call to T`.`m.

For all $g \in$ *allFieldsIn*(T):

If D $<$ T $\wedge$ `this.`g $\notin_{\mathsf{a}}$ *assigns*(T, m) $\wedge$ g is a model field,

then S<this.*g*> = S′<this.*g*>.

Proof:

The value of a model field this.*g* depends on the values of the fields accessed in its represents clause. What we have to show is that none of the concrete fields that this.*g* depends on are assignable in T.*m* when this.*g* $\notin_a$ *assigns*(T, *m*). The variable references allowed in the right side of the represents clause must have the form this.*f* or this.*x*.*y* (see the syntax of the represents clause in Figure 4.6). Furthermore, *f* can be a model field or a concrete field, but *x* must be a concrete field and *y* a model field (see the restrictions in Figure 3.10 and explained in subsections 3.3.3 and 3.3.4). Therefore, suppose this.*f* and this.*x*.*y* are accessed by the represents clause of this.*g*. It is suffices to show that these fields are not assignable in T.*m*.

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))

$\Rightarrow$ < all valid classes have to satisfy *validAssignable* of Figure 5.7 >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ *validAssignable*(T)

$\Rightarrow$ < definition of *validAssignable* of Figure 5.7 >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ ($\forall$ *m* $\in$ *setOfMethodsIn*(T), *g* $\in$ *allFieldsIn*(T) :
($\forall$ *vr* $\in_a$ *assigns*(T, *m*) : *vr* $\notin_a$ *datagroupOf*(T, *g*) $\vee$ this.*g* $\in_a$ *assigns*(T, *m*)))

$\Rightarrow$ < since *m* $\in$ *setOfMethodsIn*(T) and *g* $\in$ *allFieldsIn*(T) >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ ($\forall$ *vr* $\in_a$ *assigns*(T, *m*) : *vr* $\notin_a$ *datagroupOf*(T, *g*) $\vee$ this.*g* $\in_a$ *assigns*(T, *m*))

$\Rightarrow$ < all valid classes have to satisfy *validRepresents* of Figure 5.7 >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ ($\forall$ *vr* $\in_a$ *assigns*(T, *m*) : *vr* $\notin_a$ *datagroupOf*(T, *g*) $\vee$ this.*g* $\in_a$ *assigns*(T, *m*))
$\wedge$ *validRepresents*(T)

$\Rightarrow$ < definition of *validRepresents* of Figure 5.7 >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ ($\forall$ *vr* $\in_a$ *assigns*(T, *m*) : *vr* $\notin_a$ *datagroupOf*(T, *g*) $\vee$ this.*g* $\in_a$ *assigns*(T, *m*))
$\wedge$ ($\forall$ *g* $\in_a$ *setOfFieldsIn*(T) : !*isModelField*(*lookupField*(T, *g*)) $\vee$ *admissibleRep*(T, *g*))

$\Rightarrow$ < since *g* $\in_a$ *setOfFieldsIn*(T) and *g* is a model field >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ ($\forall$ *vr* $\in_a$ *assigns*(T, *m*) : *vr* $\notin_a$ *datagroupOf*(T, *g*) $\vee$ this.*g* $\in_a$ *assigns*(T, *m*))
$\wedge$ *admissibleRep*(T, *g*)

$\Rightarrow$ < definition of *admissibleRep* of Figure 5.7 >

this.*g* $\notin_a$ *assigns*(T, *m*) $\wedge$ this.*x*.*y*, this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)))
$\wedge$ ($\forall$ *vr* $\in_a$ *assigns*(T, *m*) : *vr* $\notin_a$ *datagroupOf*(T, *g*) $\vee$ this.*g* $\in_a$ *assigns*(T, *m*))
$\wedge$ ($\forall$ this.*f* $\in$ *accessed*(*repOf*(*lookupField*(T, *g*)) : *g* $\in$ *inOf*(*lookupField*(T, *f*)))

$\wedge$ ($\forall$ this.$f.x \in$ *accessed*(*repOf*(*lookupField*(T, g)) : ($f.x$, g) $\in$ *mapsOf*(*lookupField*(T, f)))

$\Rightarrow$ < since this.$x.y$, this.$f \in$ *accessed*(*repOf*(*lookupField*(T, g))), by the last two conjuncts and the definition of *datagroupOf* in Figure 5.6 >

this.$g \notin_a$ *assigns*(T, m) $\wedge$ this.$x.y$, this.$f \in$ *accessed*(*repOf*(*lookupField*(T, g)))

$\wedge$ ($\forall$ $vr \in_a$ *assigns*(T, m) : $vr \notin_a$ *datagroupOf*(T, g) $\vee$ this.$g \in_a$ *assigns*(T, m))

$\wedge$ this.$x.y$, this.$f \in_a$ *datagroupOf*(T, g)

$\Rightarrow$ < the second conjunct is false unless this.$x.y$, this.$f \notin_a$ *assigns*(T, m) >

this.$g \notin_a$ *assigns*(T, m) $\wedge$ this.$x.y$, this.$f \in$ *accessed*(*repOf*(*lookupField*(T, g)))

$\wedge$ this.$x.y$, this.$f \notin_a$ *assigns*(T, m)

$\wedge$ this.$x.y$, this.$f \in_a$ *datagroupOf*(T, g)

$\Rightarrow$ < logic >

this.$x.y$, this.$f \notin_a$ *assigns*(T, m)

■

Lemma 5.42 (Assignable Parameter Fields): Let D, T $\in$ *TypeId* be valid classes allowed by the rules of our technique and let m be a method declared in T. Let S $\in$ *State* be the pre-state prior to the execution of T.m and let S′ $\in$ *State* be the post-state after the execution of T.m. Let D be the run-time type of the receiver of a call to T.m.

For all $g \in$ *allFieldsIn*(T):

If D < T $\wedge$ p.$g \notin_a$ *assigns*(T, m) $\wedge$ g is a model field,

then S<p.g> = S′<p.g>.

Proof:

The proof will be by contradiction, i.e., we will prove that no valid call chain can violate this lemma. We can reason, without loss of generality, about call chains that start with the call of T.m and contain only one object-call on the formal parameter p since any valid call chain that changes the state of p must have such a suffix. Therefore, the call chain will start with a call of T.m and end with an object-call on p to a method that changes g.

We now prove that the premise of the lemma p.$g \notin_a$ *assigns*(T, m) does not hold when the conclusion is false. We start calculating from the negation of the lemma's conclusion.

S<p.g> $\neq$ S′<p.g>

$\Rightarrow$ < since S<p.g> $\neq$ S′<p.g>, there must be an object-call on p, at the end of a chain, to a method U.n such that U.n has permission to assign to this.g (from the Assignable Model Fields Lemma 5.41) >

S<p.g> $\neq$ S′<p.g> $\wedge$ U.n is an object-call on p

$\wedge$ <T.m, T1.*m1*, ... , Tk.*mk*, U.n> is a valid call chain

$\wedge$ this.$g \in_a$ *assigns*(U, n) $\wedge$ *selfAssigns*(p, U, n) $\subseteq_a$ *assigns*(Tk, *mk*)

$\Rightarrow$ < in order for the parameter in Tk.*mk* to reference the same object as in T.m, p must be passed to each call between T1.*m1* and Tk.*mk*; therefore, by the

Parameter Assignable Lemma 5.37 >

S<p.*g*> ≠ S′<p.*g*> ∧ U.*n* is an object-call on p

∧ <T.*m*, T1.*m1*, ... , Tk.*mk*, U.*n*> is a valid call chain

∧ this.*g* ∈$_a$ *assigns*(U, *n*) ∧ *selfAssigns*(p, U, *n*) ⊆$_a$ *assigns*(Tk, *mk*)

∧ *parmAssigns*(p, Tk, *mk*) ⊆$_a$ *assigns*(T, *m*)

⇒ < by Parameter Field Assignments Lemma 5.27 >

S<p.*g*> ≠ S′<p.*g*> ∧ U.*n* is an object-call on p

∧ <T.*m*, T1.*m1*, ... , Tk.*mk*, U.*n*> is a valid call chain

∧ this.*g* ∈$_a$ *assigns*(U, *n*) ∧ *selfAssigns*(p, U, *n*) ⊆$_a$ *parmAssigns*(p, Tk, *mk*)

∧ *parmAssigns*(p, Tk, *mk*) ⊆$_a$ *assigns*(T, *m*)

⇒ < since all the p's reference the same object and by the transitivity of ⊆$_a$ >

S<p.*g*> ≠ S′<p.*g*> ∧ U.*n* is an object-call on p

∧ <T.*m*, T1.*m1*, ... , Tk.*mk*, U.*n*> is a valid call chain

∧ this.*g* ∈$_a$ *assigns*(U, *n*) ∧ *selfAssigns*(p, U, *n*) ⊆$_a$ *assigns*(T, *m*)

⇒ < by Self Assignments Lemma 5.26 >

S<p.*g*> ≠ S′<p.*g*> ∧ U.*n* is an object-call on p

∧ <T.*m*, T1.*m1*, ... , Tk.*mk*, U.*n*> is a valid call chain

∧ this.*g* ∈$_a$ *selfAssigns*(this, U, *n*) ∧ *selfAssigns*(p, U, *n*) ⊆$_a$ *assigns*(T, *m*)

⇒ < since the p in T.*m* and this in U.*n* reference the same object >

S<p.*g*> ≠ S′<p.*g*> ∧ p.*g* ∈$_a$ *assigns*(T, *m*)

⇒ < the second conjunct contradicts the assumption that p.*g* ∉$_a$ *assigns*(T, *m*)
from the premise of the lemma >

false

So the conclusion of the lemma must hold.

■

Theorem 5.43 (Assignable Clause): Let D, T ∈ *TypeId* be valid classes allowed by the rules of our technique and let *m* be a method declared in T. Let S ∈ *State* be the pre-state prior to the execution of T.*m* and let S′∈ *State* be the post-state after the execution of T.*m*. Let D be the run-time type of the receiver of a call to T.*m*. Let *vr* be a directly or indirectly declared field accessible in class T's specification or in T.*m*'s method body.

If *vr* ∉$_a$ *assigns*(T, *m*),

then S<*vr*> = S′<*vr*>.

Proof:

This theorem follows from the Assignable Concrete Fields Lemma 5.39, the Assignable Pivot Objects Lemma 5.40, the Assignable Model Fields Lemma 5.41, and the Assignable Parameter Fields Lemma 5.42 since these lemmas cover the possible accessible fields (*vr*) that are assignable in T.*m*.

■

5.2.5 Additional Side-Effects Theorem

In this subsection, we prove that our technique does not allow calls to superclass methods that may make downcalls to methods with additional side-effects, i.e., methods with permission to change subclass fields. As explained previously in subsection 2.2.3, without super-class code, it is not possible to reason about the state of these subclass fields after such super-calls. Furthermore, the super-class specification cannot say anything about the state of these subclass fields since subclass fields are not in scope.

Lemma 5.44 (Subclass Side-Effects): Let *rcvr*.*m*(*e*) be a self-call or super-call allowed by our technique in some method U.*n*. Let T1, T, U ∈ *TypeId* be valid classes allowed by the rules of our technique.

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 < U ∧ *noAddSideEffects*(T1, U, *n*),
then *noAddSideEffects*(T1, T, *m*).
Proof:

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 < U ∧ *noAddSideEffects*(T1, U, *n*)
⇒ < logic >
noAddSideEffects(T1, U, *n*)
⇒ < definition of *noAddSideEffects* in Figure 5.5 >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
this.*f* ∉$_a$ *assigns*(U, *n*) ∧ this.*f*.*g* ∉$_a$ *assigns*(U, *n*))
⇒ < from the T-Call and T-SupCall rules of Figure 5.1 and since the call is made from U.*n* >
selfAssigns(this, T, *m*) ⊆$_a$ *assigns*(U, *n*)
∧ (∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
this.*f* ∉$_a$ *assigns*(U, *n*) ∧ this.*f*.*g* ∉$_a$ *assigns*(U, *n*))
⇒ < from the definition of ∈$_a$ and ⊆$_a$ >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
this.*f* ∉$_a$ *selfAssigns*(this, T, *m*) ∧ this.*f*.*g* ∉$_a$ *selfAssigns*(this, T, *m*))
⇒ < by the Self Assignments Lemma 5.26 >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
this.*f* ∉$_a$ *assigns*(T, *m*) ∧ this.*f*.*g* ∉$_a$ *assigns*(T, *m*))
⇒ < definition of *noAddSideEffects* in Figure 5.5 >
noAddSideEffects(T1, T, *m*)

■

Lemma 5.45 (Self-Call Side-Effects): Let this.*m*(*e*) be a self-call allowed by our technique in some method U.*n*. Let S ∈ *State* be the pre-state prior to the execution of this.*m*(*e*) and let S′ ∈ *State* be the post-state after the execution of this.*m*(*e*). Let T, T1 ∈ *TypeId* be valid classes allowed by the rules of our technique.

If T = *whereMethodDecl*(U, *m*) ∧ T1 ≤ U ∧ *noAddSideEffects*(T1, T, *m*),
then (∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
S<`this`.*f*> = S′<`this`.*f*> ∧ S<`this`.*f*.*g*> = S′<`this`.*f*.*g*>)

Proof:

T1 < T ∧ *noAddSideEffects*(T1, T, *m*)
⇒ < definition of *noAddSideEffects* in Figure 5.5 >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
`this`.*f* ∉$_a$ *assigns*(T, *m*) ∧ `this`.*f*.*g* ∉$_a$ *assigns*(T, *m*))
⇒ < by the Assignable Clause Theorem 5.43 >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
S<`this`.*f*> = S′<`this`.*f*> ∧ S<`this`.*f*.*g*> = S′<`this`.*f*.*g*>)

■

Lemma 5.46 (Object-Call Side-Effects): Let T1, T, U ∈ *TypeId* be valid classes allowed by the rules of our technique. Let *rcvr*.*m*(*e*) be an object-call allowed by our technique in some method U.*n*. Let S ∈ *State* be the pre-state prior to the execution of *rcvr*.*m*(*e*) and let S′ ∈ *State* be the post-state after the execution of *rcvr*.*m*(*e*).

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 ≤ U
∧ *noParmAddSideEffects*(T1, U, *n*, T, *m*),
then (∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
S<`this`.*f*> = S′<`this`.*f*> ∧ S<`this`.*f*.*g*> = S′<`this`.*f*.*g*>)

Proof:

If *e* is not `this`, then, by the Self Callback Lemma 5.21, the object-call cannot modify the state of the receiver `this`. Therefore, we have to show that the lemma holds when *e* ≡ `this`.

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 ≤ U
∧ *noParmAddSideEffects*(T1, U, *n*, T, *m*) ∧ *e* ≡ `this`
⇒ < *rcvr*.*m*(*e*) must satisfy predicate *parmAssigns* for *e* ≡ `this` by the T-Call rule of Figure 5.1 >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 ≤ U
∧ *noParmAddSideEffects*(T1, U, *n*, T, *m*)
∧ *parmAssigns*(`this`, T, *m*) ⊆$_a$ *assigns*(U, *n*)
⇒ < definition of *noParmAddSideEffects* in Figure 5.5 >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 ≤ U
∧ (*getParmType*(T, *m*) ∉ *TypeId* ∨ !(U ≤ *getParmType*(T, *m*))
∨ !(*parmAssigns*(`this`, T, *m*) ⊆$_a$ *assigns*(U, *n*))
∨ (∀ *f* ∈ *setOfFieldsIn*(T1) : `p`.*f* ∉$_a$ *assigns*(T, *m*)))
∧ *parmAssigns*(`this`, T, *m*) ⊆$_a$ *assigns*(U, *n*)
⇒ < logic >

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 ≤ U
∧ (*getParmType*(T, *m*) ∉ *TypeId* ∨ !(U ≤ *getParmType*(T, *m*))
∨ (∀ *f* ∈ *setOfFieldsIn*(T1) : `p`.*f* ∉$_a$ *assigns*(T, *m*)))
∧ *parmAssigns*(`this`, T, *m*) ⊆$_a$ *assigns*(U, *n*)
⇒ < if `this` can be passed as the formal parameter in *rcvr*.*m*(`this`), then the two disjuncts about the type of the formal parameter of T.*m* must be false >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T1 ≤ U
∧ (∀ *f* ∈ *setOfFieldsIn*(T1) : `p`.*f* ∉$_a$ *assigns*(T, *m*))
∧ *parmAssigns*(`this`, T, *m*) ⊆$_a$ *assigns*(U, *n*)
⇒ < by the Parameter Field Assignments Lemma 5.27 >
(∀ *f* ∈ *setOfFieldsIn*(T1) : `p`.*f* ∉$_a$ *parmAssigns*(`p`, T, *m*))
∧ *parmAssigns*(`this`, T, *m*) ⊆$_a$ *assigns*(U, *n*)
⇒ < by the Assignable Parameter Fields Lemma 5.42, `p`.*f* does not change during the execution of T.*m*, and, by the Assignable Model Fields Lemma 5.41, all `p`.*f*.*g* do not change since *f*.*g* would have to be a member of an assignable data group of T1 and there are no such data groups by the first conjunct; so, by the second conjunct, `p` is an alias for `this` in the pre-state and post-state of the call of T.*m* >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
`this`.*f* ∉$_a$ *assigns*(U, *n*) ∧ `this`.*f*.*g* ∉$_a$ *assigns*(U, *n*))
⇒ < by the Assignable Clause Theorem 5.43 >
(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
S<`this`.*f*> = S′<`this`.*f*> ∧ S<`this`.*f*.*g*> = S′<`this`.*f*.*g*>)
■

Lemma 5.47 (Valid Calls): Let T, T1 ∈ *TypeId* be valid classes allowed by the rules of our technique and let *m* be a method declared in T. Let S ∈ *State* be the pre-state prior to the execution of T.*m* and let S′ ∈ *State* be the post-state after the execution of T.*m*.

If T1 ≤ T ∧ *validCalls*(T1, T, *m*),
then (∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :
S<`this`.*f*> = S′<`this`.*f*> ∧ S<`this`.*f*.*g*> = S′<`this`.*f*.*g*>)

Proof:

The proof will be by induction on the number of calls in a valid call chain following the call of T.*m*.

Basis: *k* = 0, i.e., T.*m* makes no method calls.

Fields declared in T1 are not in scope since T1 < T; thus assignments to these fields and fields of objects referenced by pivot fields declared in T1 cannot be initiated by T.*m*. Also, by the Actual Parameter Aliasing Lemma 5.24, the formal parameter cannot be an alias of the receiver or reference a pivot object of the receiver. Therefore, the only way T.*m* can change the state of a subclass field or

object referenced by a subclass field is through a downcall; however, since T.*m* makes no calls, T.*m* cannot change fields in T1. Therefore,

$(\forall f \in setOfFieldsIn(T1), g \in VarId :$
$S<\texttt{this}.f> = S'<\texttt{this}.f> \wedge S<\texttt{this}.f.g> = S'<\texttt{this}.f.g>)$

Induction Step: Let k = N.

The induction hypothesis asserts that

If $T1 \leq T \wedge validCalls(T1, T, m)$
$\wedge\ k < N \wedge <T.m, U1.n1, U2.n2, \ldots, Uk.nk>$ is a valid call chain
then $(\forall f \in setOfFieldsIn(T1), g \in VarId :$
$S<\texttt{this}.f> = S'<\texttt{this}.f> \wedge S<\texttt{this}.f.g> = S'<\texttt{this}.f.g>)$

$T1 < T \wedge validCalls(T1, T, m)$
$\wedge\ k = N \wedge <T.m, U1.n1, U2.n2, \ldots, Uk.nk>$ is a valid call chain
$\Rightarrow <$ definition of *validCalls* in Figure 5.5 $>$
$T1 < T \wedge validSelfCalls(T1, T, m) \wedge validObjectCalls(T1, T, m)$
$\wedge (\forall U::n \in calls(T, m) : validCalls(T1, U, n))$
$\wedge\ k = N \wedge <T.m, U1.n1, U2.n2, \ldots, Uk.nk>$ is a valid call chain
$\Rightarrow <$ definition of *validSelfCalls* of Figure 5.5 $>$
$T1 < T \wedge (\forall \texttt{this}.n \in calls(T, m) : noAddSideEffects(T1, T, n))$
$\wedge\ validObjectCalls(T1, T, m)$
$\wedge (\forall U::n \in calls(T, m) : validCalls(T1, U, n))$
$\wedge\ k = N \wedge <T.m, U1.n1, U2.n2, \ldots, Uk.nk>$ is a valid call chain
$\Rightarrow <$ definition of *validObjectCalls* of Figure 5.5 $>$
$T1 < T \wedge (\forall \texttt{this}.n \in calls(T, m) : noAddSideEffects(T1, T, n))$
$\wedge (\forall U.n \in calls(T, m) : noParmAddSideEffects(T1, T, m, U, n))$
$\wedge (\forall U::n \in calls(T, m) : validCalls(T1, U, n))$
$\wedge\ k = N \wedge <T.m, U1.n1, U2.n2, \ldots, Uk.nk>$ is a valid call chain
$\Rightarrow <$ none of the self-calls can change fields in T1 (by the Self-Call Side-Effects Lemma 5.45), none of the object-calls can change fields in T1 (by the Object-Call Side-Effects Lemma 5.45), and none of the super-calls can change fields in T1 (by the induction hypothesis) $>$
$(\forall f \in setOfFieldsIn(T1), g \in VarId :$
$S<\texttt{this}.f> = S'<\texttt{this}.f> \wedge S<\texttt{this}.f.g> = S'<\texttt{this}.f.g>)$

■

Our next theorem proves that if our technique allows a superclass method to be called, then that method does not at any time during execution modify fields of its run-time subclasses. However, note that it does not say that the overriding method does not have permission to change subclass fields, but

rather it says that our technique only allows a superclass method to be called, when that superclass method does not make downcalls that modify subclass fields. This is important when superclass code is unavailable as explained in subsection 2.2.3.

Theorem 5.48 (Additional Side-Effects): Let D, T, T1 ∈ *TypeId* be valid classes allowed by the rules of our technique. Let `super.`*m*(*e*) be a super-call allowed by our technique in some method U`.`*n*. Let D be the run-time type of the current receiver. Let S ∈ *State* be the pre-state prior to the execution of `super.`*m*(*e*) and let S′ ∈ *State* be an intermediate state during the execution of `super.`*m*(*e*).

If T = *whereMethodDecl*(*superOf*(U), *m*) ∧ D ≤ T1 < T,

then (∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :

S<`this.`*f*> = S′<`this.`*f*> ∧ S<`this.`*f*.*g*> = S′<`this.`*f*.*g*>)).

Proof:

The conclusion holds when T`.`*m* has no side-effects, so the proof will assume that T`.`*m* has side-effects. Also, since any chain that changes the state of an object must begin with an object-call, we will assume, without loss of generality, that U`.`*n* is an object-call (or can be self-called).

There are two possibilities, either method T`.`*m* is overridden or T`.`*m* is not overridden in any subclass of T. We will divide the proof into four cases, one case for when T`.`*m* is not overridden and three cases for when T`.`*m* is overridden depending where T1 is in the hierarchy in relation to U.

Case 1: method T`.`*m* is not overridden in any subclass of T

T = *whereMethodDecl*(*superOf*(U), *m*) ∧ D ≤ T1 < T ∧ ! *isOverridden*(T1, *m*)

⇒ < since T`.`*m* is not overridden in any subclass of T >

T = *whereMethodDecl*(*superOf*(U), *m*) ∧ D ≤ T1 < T ∧ ! *isOverridden*(T1, *m*)

∧ T`.`*m* ∈ *allMethodsIn*(*superOf*(T1))

⇒ < from the Method Overriding Lemma 5.5 >

noAddSideEffects(T1, T, *m*) ∧ *validInvariant*(T1, T, *m*)

⇒ < logic >

noAddSideEffects(T1, T, *m*)

⇒ < definition of *noAddSideEffects* in Figure 5.5 >

(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :

`this.`*f* $\notin_a$ *assigns*(T, *m*) ∧ `this.`*f*.*g* $\notin_a$ *assigns*(T, *m*))

⇒ < by the Assignable Clause Theorem 5.43 >

(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :

S<`this.`*f*> = S′<`this.`*f*> ∧ S<`this.`*f*.*g*> = S′<`this.`*f*.*g*>)

Case 2: T`.`*m* is overridden in some subclass of T ∧ D ≤ T1 < U < T

D ≤ T1 < U < T

⇒ < the ability to directly object-call U`.`*n*, when D < U, means that U`.`*n* was not overridden in any subclass of U >

$D \le T1 < U < T \ \wedge \ !\ \mathit{isOverridden}(T1, n)$

$\Rightarrow <$ since $U.n$ is not overridden in any subclass of U and by the definition of *allMethodsIn* of Figure 5.6 $>$

$D \le T1 < U < T \ \wedge \ !\ \mathit{isOverridden}(T1, n) \ \wedge \ U.n \in \mathit{allMethodsIn}(\mathit{superOf}(T1))$

$\Rightarrow <$ from the Method Overriding Lemma 5.5 $>$

$D \le T1 < U < T \ \wedge \ \mathit{noAddSideEffects}(T1, U, n) \ \wedge \ \mathit{validInvariant}(T1, U, n)$

$\Rightarrow <$ logic $>$

$D \le T1 < U < T \ \wedge \ \mathit{noAddSideEffects}(T1, U, n)$

$\Rightarrow <$ since, from the premise of the lemma, `super.`$m(e)$ invokes $T{::}m$ from $U.n$ $>$

$T = \mathit{whereMethodDecl}(\mathit{superOf}(U), m)$

$\wedge \ D \le T1 < U < T \ \wedge \ \mathit{noAddSideEffects}(T1, U, n)$

$\Rightarrow <$ by the Subclass Side-Effects Lemma 5.44 $>$

$\mathit{noAddSideEffects}(T1, T, m)$

$\Rightarrow <$ definition of *noAddSideEffects* in Figure 5.5 $>$

$(\forall f \in \mathit{setOfFieldsIn}(T1), g \in \mathit{VarId}$:

$\texttt{this}.f \not\in_a \mathit{assigns}(T, m) \ \wedge \ \texttt{this}.f.g \not\in_a \mathit{assigns}(T, m)$)

$\Rightarrow <$ by the Assignable Clause Theorem 5.43 $>$

$(\forall f \in \mathit{setOfFieldsIn}(T1), g \in \mathit{VarId}$:

$S\langle\texttt{this}.f\rangle = S'\langle\texttt{this}.f\rangle \wedge S\langle\texttt{this}.f.g\rangle = S'\langle\texttt{this}.f.g\rangle$)

Case 3: $T.m$ is overridden in some subclass of $T \ \wedge \ D \le U < T1 < T$

$D \le U < T1 < T$

$\Rightarrow <$ the ability to directly super-call $T.m$ from $U.n$, when $U < T$, means that $T.m$ was not overridden in any classes in the hierarchy between U and T, e.g., in T1 $>$

$D \le U < T1 < T \ \wedge \ !\ \mathit{isOverridden}(T1, m)$

$\Rightarrow <$ since $T.m$ is not overridden in any classes between U and T and by the definition of *allMethodsIn* of Figure 5.6 $>$

$D \le U < T1 < T \ \wedge \ !\ \mathit{isOverridden}(T1, m) \ \wedge \ T.m \in \mathit{allMethodsIn}(\mathit{superOf}(T1))$

$\Rightarrow <$ from the Method Overriding Lemma 5.5 $>$

$\mathit{noAddSideEffects}(T1, T, m) \ \wedge \ \mathit{validInvariant}(T1, T, m)$

$\Rightarrow <$ logic $>$

$\mathit{noAddSideEffects}(T1, T, m)$

$\Rightarrow <$ definition of *noAddSideEffects* in Figure 5.5 $>$

$(\forall f \in \mathit{setOfFieldsIn}(T1), g \in \mathit{VarId}$:

$\texttt{this}.f \not\in_a \mathit{assigns}(T, m) \ \wedge \ \texttt{this}.f.g \not\in_a \mathit{assigns}(T, m)$)

$\Rightarrow <$ by the Assignable Clause Theorem 5.43 $>$

$(\forall f \in \mathit{setOfFieldsIn}(T1), g \in \mathit{VarId}$:

$S\langle\texttt{this}.f\rangle = S'\langle\texttt{this}.f\rangle \wedge S\langle\texttt{this}.f.g\rangle = S'\langle\texttt{this}.f.g\rangle$)

Case 4: T . *m* is overridden in some subclass of T ∧ T1 ≡ U ∧ D ≤ T1 < T

The super-call of T**::***m* is made from U . *n* so the call must satisfy the *okToSuperCall* predicate since that predicate is an antecedent in the T-SupCall rule of Figure 5.1; so we start calculating from *okToSuperCall*(U, T, *m*) since U is the static type of the receiver in U . *n*.

okToSuperCall(U, T, *m*) ∧ T1 ≡ U ∧ D ≤ T1 < T

⇒ < substituting T1 for U >

okToSuperCall(T1, T, *m*) ∧ D ≤ T1 < T

⇒ < definition of *okToSuperCall* in Figure 5.4 >

validInvariant(T1, T, *m*) ∧ *validCalls*(T1, T, *m*) ∧ D ≤ T1 < T

⇒ < logic >

validCalls(T1, T, *m*) ∧ D ≤ T1 < T

⇒ < by the Valid Calls Lemma 5.47 >

(∀ *f* ∈ *setOfFieldsIn*(T1), *g* ∈ *VarId* :

S<this . *f*> = S′<this . *f*> ∧ S<this . *f* . *g*> = S′<this . *f* . *g*>)

■

5.2.6 Validity of Our Axioms and Inference Rules

In this subsection, we prove that the axioms of our verification logic are valid and that our inference rules preserve validity.

5.2.6.1 Validity defined

We say, for a given program, that an assertion **A** is *valid*, written **|= A**, if **A** is true for all program states. Similarly, we say that an assertion **A** is *derivable* in a deductive system, written **|- A**, if **A** is derivable from the axioms of that deductive system using its axioms and inference rules. An inference rule *preserves validity* if assuming the antecedents are valid, then the consequent is also valid. Axioms must always be valid.

A Hoare triple {**P**} *C* {**Q**} is valid if and only if

(∀ S ∈ *State* : **if** S<**P**> ∧ {*C*, S} ⇒$_s$ S′, **then** S′<**Q**>)

5.2.6.2 Validity of our axioms

To prove that our programming logic is sound, we must prove that all axioms are valid and that all inference rules preserve validity. Lemmas 4.1 - 4.4 prove that our axioms are valid.

Lemma 5.49 (A-Skip): |= {P} ; {P}, i.e., the A-Skip axiom is valid

Proof:

Let S ∈ *State*. If S<**P**> = false or { ; , S} does not terminate, then the A-Skip axiom is trivially valid. So assume S<**P**> = true and that { ; , S} terminates. We start calculating with the application of the S-Skip rule of Figure 4.17.

S<**P**> ∧ { ; , S} ⇒$_s$ S

$\Rightarrow$ < logic >

S<**P**>

■

Lemma 5.50 (A-ExpAssign): |= {P[vr←*e*]} *vr*=*e*; {P}, i.e., the A-ExpAssign axiom is valid

Let S ∈ *State*. If S<**P**[vr←*e*]> = false or {*vr*=*e*;, S} does not terminate, then the A-ExpAssign axiom is trivially valid. So assume S<**P**[vr←*e*]> = true and that {*vr*=*e*;, S} terminates. We start calculating with the application of the S-ExpAssign rule of Figure 4.19.

S<**P**[vr←*e*]> ∧ [*e*, S] $\Rightarrow_e$ v ∧ [*vr*, S] $\Rightarrow_{lv}$ vLoc ∧ {*vr*=*e*;, S} $\Rightarrow_s$ S[vLoc := v]

$\Rightarrow$ < Substitution Theorem 5.9 >

S<**P**[vr←*e*]> = S[vLoc := v]<**P**> ∧ S<**P**[vr←*e*]> ∧ {*vr*=*e*;, S} $\Rightarrow_s$ S[vLoc := v]

$\Rightarrow$ < S<**P**[vr←*e*]> = true >

S[vLoc := v]<**P**>

■

Lemma 5.51 (A-LocalDecl): |= {P[x←*default*(T)]} T x; {P} , i.e., the A-LocalDecl axiom is valid

Proof:

Let S ∈ *State*. If S<**P**[x←*default*(T)]> = false or {T x;, S} does not terminate, then the A-LocalDecl axiom is trivially valid. So assume S<**P**[x←*default*(T)]> = true and that {T x;, S} terminates. We start calculating with the application of the S-LocalDecl rule of Figure 4.19.

S<**P**[x←*default*(T)]> ∧ v = *mkVal*(*default*(T)) ∧ vLoc = *loc*(x, local)

∧ {T x;, S} $\Rightarrow_s$ S[vLoc := v]

$\Rightarrow$ < *default*(T) is a literal and E-Literal rule of Figure 4.16 >

[*default*(T), S] $\Rightarrow_e$ v ∧ S<**P**[x←*default*(T)]> ∧ vLoc = *loc*(x, local)

∧ {T x;, S} $\Rightarrow_s$ S[vLoc := v]

$\Rightarrow$ < L-VarId rule of Figure 4.16 >

[*default*(T), S] $\Rightarrow_e$ v ∧ S<**P**[x←*default*(T)]> ∧ [x, S] $\Rightarrow_{lv}$ vLoc

∧ {T x;, S} $\Rightarrow_s$ S[vLoc := v]

$\Rightarrow$ < Substitution Theorem 5.9 >

S<**P**[x←*default*(T)]> = S[vLoc := v]<**P**> ∧ S<**P**[x←*default*(T)]>

∧ {T x;, S} $\Rightarrow_s$ S[vLoc := v]

$\Rightarrow$ < S<**P**[x←*default*(T)]> = true >

S[vLoc := v]<**P**>

■

Lemma 5.52 (A-Return): |= {P[\result←*e*]} return *e*; {P}, i.e., the A-Return axiom is valid

Proof:

Let S ∈ *State*. If S<**P**[\result←*e*]> = false or {`return` *e*`;`, S} does not terminate, then the A-Return axiom is trivially valid. So assume S<**P**[\result←*e*]> = true and that {`return` *e*`;`, S} terminates. We start calculating with the application of the S-Return rule of Figure 4.19.

S<**P**[\result←*e*]> ∧ [*e*, S] ⇒$_e$ v ∧ {`return` *e*`;`, S} ⇒$_s$ S[`resultLoc :=` v]

⇒ < L-Result rule of Figure 4.16 >

[\result, S] ⇒$_{lv}$ `resultLoc` ∧ S<**P**[\result←*e*]> ∧ [*e*, S] ⇒$_e$ v

∧ {`return` *e*`;`, S} ⇒$_s$ S[`resultLoc :=` v]

⇒ < Substitution Theorem 5.9 >

S<**P**[\result←*e*]> = S[`resultLoc :=` v]<**P**> ∧ S<**P**[\result←*e*]>

∧ {`return` *e*`;`, S} ⇒$_s$ S[`resultLoc :=` v]

⇒ < S<**P**[\result←*e*]> = true >

S[`resultLoc :=` v]<**P**>

■

5.2.6.3 The inference rules preserve validity

Lemma 5.53 (A-If): The A-If rule of Figure 4.20 preserves validity

Proof:

Let S ∈ *State*. If S<**P**> = false or {`if (`*e*`)` *C1* `else` *C2*, S} does not terminate, then the A-If rule is trivially valid. So assume S<**P**> = true and {`if (`*e*`)` *C1* `else` *C2*, S} terminates. Since there are two rules in the operational semantics for the if-statement, the proof requires two cases.

Case 1: suppose the application of the S-IfThen rule of Figure 4.17 terminates, i.e., S<*e*> = true

S<**P**> ∧ S<*e*> ∧ {*C1*, S} ⇒$_s$ S′ ∧ {`if (`*e*`)` *C1* `else` *C2*, S} ⇒$_s$ S′

⇒ < S<**P**> && S<*e*> = S<**P** && *e*> >

S<**P**> ∧ S<**P** && *e*> ∧ {*C1*, S} ⇒$_s$ S′ ∧ {`if (`*e*`)` *C1* `else` *C2*, S} ⇒$_s$ S′

⇒ < |= {**P** && *e*} *C1* {**Q**}, since its an antecedent of the A-If rule of Figure 4.20 >

S′<**Q**>

Case 2: suppose the application of the S-IfElse rule of Figure 4.17 terminates, i.e., S<*e*> = false

S<**P**> ∧ S<!*e*> ∧ {*C2*, S} ⇒$_s$ S′ ∧ {`if (`*e*`)` *C1* `else` *C2*, S} ⇒$_s$ S′

⇒ < S<**P**> && S<!*e*> = S<**P** && !*e*> >

S<**P**> ∧ S<**P** && !*e*> ∧ {*C2*, S} ⇒$_s$ S′ ∧ {`if (`*e*`)` *C1* `else` *C2*, S} ⇒$_s$ S′

⇒ < |= {**P** && !*e*} *C2* {**Q**}, since its an antecedent of the A-If rule of Figure 4.20 >

S′<**Q**>

Therefore, the A-If rule preserves validity whether the condition S<*e*> is true or false.

■

Lemma 5.54 (A-Seq): The A-Seq rule of Figure 4.20 preserves validity

Proof:

Let $\mathsf{S} \in \mathit{State}$. If $\mathsf{S}\langle\mathbf{P}\rangle$ = false or $\{\mathit{C1}\ \mathit{C2}, \mathsf{S}\}$ does not terminate, then the A-Seq rule is trivially valid. So assume $\mathsf{S}\langle\mathbf{P}\rangle$ = true and $\{\mathit{C1}\ \mathit{C2}, \mathsf{S}\}$ terminates.

$\mathsf{S}\langle\mathbf{P}\rangle \wedge \{\mathit{C1}\ \mathit{C2}, \mathsf{S}\} \Rightarrow_s \mathsf{S}''$

$\Rightarrow$ < by the S-Seq rule of Figure 4.17 >

$\mathsf{S}\langle\mathbf{P}\rangle \wedge \{\mathit{C1}, \mathsf{S}\} \Rightarrow_s \mathsf{S}' \wedge \{\mathit{C2}, \mathsf{S}'\} \Rightarrow_s \mathsf{S}'' \wedge \{\mathit{C1}\ \mathit{C2}, \mathsf{S}\} \Rightarrow_s \mathsf{S}''$

$\Rightarrow$ < $\models \{\mathbf{P}\}\ \mathit{C1}\ \{\mathbf{Q}\}$, since its an antecedent of the A-Seq rule of Figure 4.20 >

$\mathsf{S}'\langle\mathbf{Q}\rangle \wedge \{\mathit{C2}, \mathsf{S}'\} \Rightarrow_s \mathsf{S}'' \wedge \{\mathit{C1}\ \mathit{C2}, \mathsf{S}\} \Rightarrow_s \mathsf{S}''$

$\Rightarrow$ < $\models \{\mathbf{Q}\}\ \mathit{C1}\ \{\mathbf{R}\}$, since its an antecedent of the A-Seq rule of Figure 4.20 >

$\mathsf{S}''\langle\mathbf{R}\rangle$

■

Lemma 5.55 (A-Conseq): The A-Conseq rule of Figure 4.20 preserves validity

Proof:

Let $\mathsf{S} \in \mathit{State}$. If $\mathsf{S}\langle\mathbf{P}\rangle$ = false or $\{\mathit{C}, \mathsf{S}\}$ does not terminate, then the A-Conseq rule is trivially valid. So assume $\mathsf{S}\langle\mathbf{P}\rangle$ = true and $\{\mathit{C}, \mathsf{S}\}$ terminates.

$\mathsf{S}\langle\mathbf{P}\rangle \wedge \{\mathit{C}, \mathsf{S}\} \Rightarrow_s \mathsf{S}'$

$\Rightarrow$ < $\models \mathbf{P} \Rightarrow \mathbf{P}'$, since its an antecedent of the A-Conseq rule of Figure 4.20 >

$\mathsf{S}\langle\mathbf{P}'\rangle \wedge \{\mathit{C}, \mathsf{S}\} \Rightarrow_s \mathsf{S}'$

$\Rightarrow$ < $\models \{\mathbf{P}'\}\ \mathit{C}\ \{\mathbf{Q}'\}$, since its an antecedent of the A-Conseq rule of Figure 4.20 >

$\mathsf{S}'\langle\mathbf{Q}'\rangle \wedge \{\mathit{C}, \mathsf{S}\} \Rightarrow_s \mathsf{S}'$

$\Rightarrow$ < $\models \mathbf{Q}' \Rightarrow \mathbf{Q}$, since its an antecedent of the A-Conseq rule of Figure 4.20 >

$\mathsf{S}'\langle\mathbf{Q}\rangle$

■

Lemma 5.56 (A-Represents): The A-ModelRep and A-ExpRep rules of Figure 4.20 preserve validity

Proof:

Let $\mathsf{S} \in \mathit{State}$. If $\mathsf{S}\langle\mathbf{P}\rangle$ = false or $\mathsf{S}\langle\mathbf{P}\rangle$ does not terminate, then the A-ModelRep and A-ExpRep rules are trivially valid. So assume $\mathsf{S}\langle\mathbf{P}\rangle$ = true and $\mathsf{S}\langle\mathbf{P}\rangle$ terminates. We start calculating from the antecedents of these rules (which are the same).

$\mathsf{S}\langle\mathbf{P}\rangle \wedge$ represents `this.`F `<-` e`;`

$\Rightarrow$ < by the semantics of the represents clause >

$\mathsf{S}\langle\mathbf{P}\rangle \wedge \mathsf{S}\langle$`this.`$F\rangle = \mathsf{S}\langle e\rangle$

$\Rightarrow$ < meaning of substitution and equality >

$\mathsf{S}\langle\mathbf{P}\rangle \wedge \mathsf{S}\langle$`this.`$F\rangle = \mathsf{S}\langle e\rangle \wedge \mathsf{S}\langle\mathbf{P}\rangle = \mathsf{S}\langle\mathbf{P}$[`this.`$F \leftarrow$ `this.`F]$\rangle$

$\Rightarrow$ < since $\mathsf{S}\langle$`this.`$F\rangle = \mathsf{S}\langle e\rangle$ and substitution of equals for equals >

$\mathsf{S}\langle\mathbf{P}\rangle \wedge \mathsf{S}\langle\mathbf{P}\rangle = \mathsf{S}\langle\mathbf{P}$[`this.`$F \leftarrow e$]$\rangle \wedge \mathsf{S}\langle\mathbf{P}\rangle = \mathsf{S}\langle\mathbf{P}$[`e`$\leftarrow$ `this.`F]$\rangle$

$\Rightarrow$ < equality >

S<**P**[`this.`$F$$\leftarrow$$e$]> $\wedge$ S<**P**[e$\leftarrow$`this.`F]>

■

Lemma 5.57 (A-While): The A-While rule of Figure 4.20 preserves validity

Proof:

Let S $\in$ *State*. If S<**P**> = false or {`while (`e`)` C, S} does not terminate, then the A-While rule is trivially valid. So assume S<**P**> = true and that {`while (`e`)` C, S} terminates. The proof will by induction on the number times the S-While rule of Figure 4.17 was applied before termination.

Basis:

Suppose the S-While rule was applied 0 times. Therefore, the S-EndWhile rule of Figure 4.17 had to be applied and S<e> = false.

S<**P**> $\wedge$ S<!e> $\wedge$ {`while (`e`)` C, S} $\Rightarrow_s$ S

$\Rightarrow$ < S<**P**> && S<!e> = S<**P** && !e> >

S<**P** && !e>

Induction Step:

The induction hypothesis says that if S<**P**> holds and {`while (`e`)` C, S} $\Rightarrow_s$ S′ terminates in fewer than N applications of the S-While rule, then S′[**P** && !e] holds. Let S be a state such that S<**P**> = true and {`while (`e`)` C, S} $\Rightarrow_s$ S′ terminates after at least 1 and less than or equal to N applications of the S-While rule.

S<**P**> $\wedge$ S<e> $\wedge$ {`while (`e`)` C, S} $\Rightarrow_s$ S′

$\Rightarrow$ < at least 1 application of the S-While rule of Figure 4.17 >

S<**P**> $\wedge$ S<e> $\wedge$ {C `while (`e`)` C, S} $\Rightarrow_s$ S′

$\Rightarrow$ < the A-Seq rule of Figure 4.17 >

S<**P**> $\wedge$ S<e> $\wedge$ {C, S} $\Rightarrow_s$ S1 $\wedge$ {`while (`e`)` C, S1} $\Rightarrow_s$ S′

$\Rightarrow$ < S<**P**> && S<e> = S<**P** && e> >

S<**P**> $\wedge$ S<**P** && e> $\wedge$ {C, S} $\Rightarrow_s$ S1 $\wedge$ {`while (`e`)` C, S1} $\Rightarrow_s$ S′

$\Rightarrow$ < |= {**P** && e} C {**P**}, since its an antecedent of the A-While rule of Figure 4.20 >

S1<**P**> $\wedge$ {`while (`e`)` C, S1} $\Rightarrow_s$ S′

$\Rightarrow$ < {`while (`e`)` C, S1} $\Rightarrow_s$ S′ finishes in fewer than N applications of S-While rule, so by I.H. >

S′<**P** && !e>

■

5.2.6.4 Eliminating \old-expressions and combining specification cases

In JML, the triple {**P**} C {**Q**} means that if {C, S} $\Rightarrow_s$ S′ terminates, then S′<**Q**> must hold. However, in JML, when **Q** contains `\old`-expressions, these `\old`-expressions must be replaced by the value of that expression when evaluated in the pre-state, i.e., **Q**[`\old(`e`)`$\leftarrow$S<e>] for all `\old(`e`)`

expressions occurring in **Q**. This meaning of `\old`-expressions is used in the proof of the next lemma; the next two lemmas allow verifiers to eliminate `\old`-expressions from the postcondition of method specifications when verifying or calling a method.

Lemma 5.58 (A-OldVerify): The A-OldVerify rule of Figure 4.23 preserves validity

Proof:

Let $S \in State$. If $S\langle \mathbf{P} \rangle$ = false or $\{C, S\}$ does not terminate, then the A-OldVerify rule is trivially valid. So assume $S\langle \mathbf{P} \rangle$ = true and $\{C, S\}$ terminates.

$S\langle \mathbf{P} \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < let Z be a logical variable such that $Z == S\langle e \rangle$ >

$S\langle \mathbf{P} \rangle \wedge Z == S\langle e \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < Z is a logical variable and thus is independent of the program state >

$S\langle \mathbf{P} \rangle \wedge S\langle Z == e \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < logic and meaning of $S\langle \mathbf{P} \;\&\&\; Z == e \rangle$ >

$S\langle \mathbf{P} \;\&\&\; Z == e \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < $\models \{\mathbf{P} \;\&\&\; Z == e\}\ C\ \{\mathbf{Q}[\texttt{\textbackslash old}(e) \leftarrow Z]\}$, since it is an antecedent of the A-OldVerify rule of Figure 4.23 >

$S'\langle \mathbf{Q}[\texttt{\textbackslash old}(e) \leftarrow Z] \rangle \wedge \{C, S\} \Rightarrow_s S' \wedge Z == S\langle e \rangle$

$\Rightarrow$ < $\texttt{\textbackslash old}(e) = S\langle e \rangle$, by the semantics of $\texttt{\textbackslash old}(e)$, and $Z == S\langle e \rangle$ >

$S'\langle \mathbf{Q}[\texttt{\textbackslash old}(e) \leftarrow \texttt{\textbackslash old}(e)] \rangle$

$\Rightarrow$ < substitution >

$S'\langle \mathbf{Q} \rangle$

■

Lemma 5.59 (A-OldCall): The A-OldCall rule of Figure 4.23 preserves validity

Proof:

Let $S \in State$. If for all $Z \in LogicId$, $S\langle \mathbf{P} \;\&\&\; Z == e \rangle$ = false or $\{C, S\}$ does not terminate, then the A-OldCall rule is trivially valid. So assume $S\langle \mathbf{P} \;\&\&\; Z == e \rangle$ = true and $\{C, S\}$ terminates.

$S\langle \mathbf{P} \;\&\&\; Z == e \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < logic and meaning of $S\langle \mathbf{P} \;\&\&\; Z == e \rangle$ >

$S\langle \mathbf{P} \rangle \wedge S\langle Z == e \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < Z is a logical variable and thus is independent of the program state >

$S\langle \mathbf{P} \rangle \wedge Z == S\langle e \rangle \wedge \{C, S\} \Rightarrow_s S'$

$\Rightarrow$ < $\models \{\mathbf{P}\}\ C\ \{\mathbf{Q}\}$, since it is an antecedent of the A-OldCall rule of Figure 4.23 >

$S'\langle \mathbf{Q} \rangle \wedge \{C, S\} \Rightarrow_s S' \wedge Z == S\langle e \rangle$

$\Rightarrow$ < the meaning of $S'\langle \mathbf{Q} \rangle$ when $\mathbf{Q}$ contains $\texttt{\textbackslash old}(e)$ >

$S'\langle \mathbf{Q}[\texttt{\textbackslash old}(e) \leftarrow S\langle e \rangle] \rangle \wedge \{C, S\} \Rightarrow_s S' \wedge Z == S\langle e \rangle$

$\Rightarrow$ < substitution and $Z == S\langle e \rangle$ >

S′<**Q**[\old(*e*)←*Z*]>

■

Lemma 5.60 (A-SpecCase): The A-SpecCase rule of Figure 4.23 preserves validity

Proof:

Let S ∈ *State*. If for all *Z*, *Z′* ∈ *LogicId*, S<(**P** || **P′**) && *Z*==**P** && *Z′*==**P′**> = false or {*C*, S} does not terminate, then the A-SpecCase rule is trivially valid. Therefore, assume {*C*, S} terminates and S<(**P** || **P′**) && *Z*==**P** && *Z′*==**P′**> = true.

Case 1: suppose S<**P**> = true and S<**P′**> = false

S<(**P** || **P′**) && *Z*==**P** && *Z′*==**P′**> ∧ S<**P**> ∧ S<**P′**> = false ∧ {*C*, S} ⇒$_s$ S′

⇒ < logic and meaning of S<*e*> >

S<(**P** || **P′**)> ∧ S<*Z*==**P**> ∧ S<*Z′*==**P′**> ∧ S<**P**> ∧ S<**P′**> = false ∧ {*C*, S} ⇒$_s$ S′

⇒ < *Z* is a logical variable and thus is independent of the program state >

Z==S<**P**> ∧ *Z′*==S<**P′**> ∧ S<**P**> ∧ S<**P′**> = false ∧ {*C*, S} ⇒$_s$ S′

⇒ < logic and substitution >

Z==true ∧ *Z′*==false ∧ S<**P**> ∧ {*C*, S} ⇒$_s$ S′

⇒ < |= {**P**} *C* {**Q**}, since it is an antecedent of the A-SpecCase rule of Figure 4.23 >

S′<**Q**> ∧ *Z*==true ∧ *Z′*==false ∧ {*C*, S} ⇒$_s$ S′

⇒ < logic and *Z* = true, *Z′* = false, and S′<true>=true >

S′<!*Z* || **Q**> ∧ S′<!*Z′* || **Q′**>

⇒ < logic and meaning of S<*e*> and && >

S′<(!*Z* || **Q**) && (!*Z′* || **Q′**)>

Case 2: suppose S<**P**> = false and S<**P′**> = true

S<(**P** || **P′**) && *Z*==**P** && *Z′*==**P′**> ∧ S<**P**> = false ∧ S<**P′**> ∧ {*C*, S} ⇒$_s$ S′

⇒ < logic and meaning of S<*e*> >

S<(**P** || **P′**)> ∧ S<*Z*==**P**> ∧ S<*Z′*==**P′**> ∧ S<**P**> = false ∧ S<**P′**> ∧ {*C*, S} ⇒$_s$ S′

⇒ < *Z* is a logical variable and thus is independent of the program state >

Z==S<**P**> ∧ *Z′*==S<**P′**> ∧ S<**P**> = false ∧ S<**P′**> ∧ {*C*, S} ⇒$_s$ S′

⇒ < logic and substitution >

Z==false ∧ *Z′*==true ∧ S<**P**> ∧ {*C*, S} ⇒$_s$ S′

⇒ < |= {**P′**} *C* {**Q′**}, since it is an antecedent of the A-SpecCase rule of Figure 4.23 >

S′<**Q′**> ∧ *Z*==false ∧ *Z′*==true ∧ {*C*, S} ⇒$_s$ S′

⇒ < logic and *Z* = false, *Z′* = true, and S′<true>=true >

S′<!*Z* || **Q**> ∧ S′<!*Z′* || **Q′**>

⇒ < logic and meaning of S<*e*> >

S′<(!*Z* || **Q**) && (!*Z′* || **Q′**)>

Case 3: suppose S<**P**> = true and S<**P′**> = true

S<(**P** || **P′**) && *Z*==**P** && *Z′*==**P′**> ∧ S<**P**> ∧ S<**P′**> ∧ {*C*, S} ⇒$_s$ S′

$\Rightarrow$ < logic and meaning of S<*e*> >

S<(**P** || **P**′)> $\wedge$ S<*Z*==**P**> $\wedge$ S<*Z*′==**P**′> $\wedge$ S<**P**> $\wedge$ S<**P**′> $\wedge$ {*C*, S} $\Rightarrow_{\mathsf{S}}$ S'

$\Rightarrow$ < *Z* is a logical variable and thus is independent of the program state >

Z==S<**P**> $\wedge$ *Z*′==S<**P**′> $\wedge$ S<**P**> $\wedge$ S<**P**′> $\wedge$ {*C*, S} $\Rightarrow_{\mathsf{S}}$ S'

$\Rightarrow$ < logic and substitution >

Z==true $\wedge$ *Z*′==true $\wedge$ S<**P**> $\wedge$ S<**P**′> $\wedge$ {*C*, S} $\Rightarrow_{\mathsf{S}}$ S'

$\Rightarrow$ < |= {**P**} *C* {**Q**}, since it is an antecedent of the A-SpecCase rule of Figure 4.23 >

S'<**Q**> $\wedge$ *Z*==true $\wedge$ *Z*′==true $\wedge$ S<**P**′> $\wedge$ {*C*, S} $\Rightarrow_{\mathsf{S}}$ S'

$\Rightarrow$ < |= {**P**′} *C* {**Q**′}, since its an antecedent of the A-SpecCase rule of Figure 4.23 >

S'<**Q**′> $\wedge$ S'<**Q**> $\wedge$ *Z*==true $\wedge$ *Z*′==true $\wedge$ {*C*, S} $\Rightarrow_{\mathsf{S}}$ S'

$\Rightarrow$ < logic and *Z* = true and *Z*′ = true>

S'<!*Z* || **Q**> $\wedge$ S'<!*Z*′ || **Q**′>

$\Rightarrow$ < logic and meaning of S<*e*> >

S'<(!*Z* || **Q**) && (!*Z*′ || **Q**′)>

■

5.2.6.5 Method and constructor correctness

In this subsection, we prove that subclasses are behavioral subtypes of their superclasses, i.e., each overriding subclass method satisfies the specification of the superclass method it overrides (see also subsection 1.4.1). We also prove that, on exit, methods establish the run-time type invariant of its argument objects (i.e., the receiver and formal parameter).

Lemma 5.61 (Subtype): Let T1, T $\in$ *TypeId* and let *m* $\in$ *MethId*.

If T1 $\leq$ T $\wedge$ *methodsOf*(`EnvT`(T1))(*m*) $\neq$`undef` $\wedge$ *methodsOf*(`EnvT`(T))(*m*) $\neq$`undef`

$\wedge$ |= {*inv*(T1) && *req*(T1, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T1, *m*)},

then |= {*inv*(T1) && *req*(T, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T, *m*)}

Proof:

T1 $\leq$ T $\wedge$ *methodsOf*(`EnvT`(T1))(*m*) $\neq$`undef` $\wedge$ *methodsOf*(`EnvT`(T))(*m*) $\neq$`undef`

$\wedge$ |= {*inv*(T1) && *req*(T1, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T1, *m*)}

$\Rightarrow$ < by the Requires Clause Lemma 5.2 >

T1 $\leq$ T $\wedge$ *methodsOf*(`EnvT`(T1))(*m*) $\neq$`undef` $\wedge$ *methodsOf*(`EnvT`(T))(*m*) $\neq$`undef`

$\wedge$ |= {*inv*(T1) && *req*(T1, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T1, *m*)}

$\wedge$ (*req*(T, *m*) $\Rightarrow$ *req*(T1, *m*))

$\Rightarrow$ < by the Ensures Clause Lemma 5.3 >

T1 $\leq$ T $\wedge$ *methodsOf*(`EnvT`(T1))(*m*) $\neq$`undef` $\wedge$ *methodsOf*(`EnvT`(T))(*m*) $\neq$`undef`

$\wedge$ |= {*inv*(T1) && *req*(T1, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T1, *m*)}

$\wedge$ (*req*(T, *m*) $\Rightarrow$ *req*(T1, *m*)) $\wedge$ (*ens*(T1, *m*) $\Rightarrow$ *ens*(T, *m*))

$\Rightarrow$ < logic >

T1 ≤ T ∧ *methodsOf*(`EnvT`(T1))(*m*) ≠ `undef` ∧ *methodsOf*(`EnvT`(T))(*m*) ≠ `undef`
∧ |= {*inv*(T1) && *req*(T1, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T1, *m*)}
∧ (*inv*(T1) && *req*(T, *m*) ⇒ *inv*(T1) && *req*(T1, *m*))
∧ (*inv*(T1) && *ens*(T1, *m*) ⇒ *inv*(T1) && *ens*(T, *m*))
⇒ < by the A-Conseq rule which preserves validity from Lemma 5.55 >
|= {*inv*(T1) && *req*(T, *m*)} *getBody*(T1, *m*) {*inv*(T1) && *ens*(T, *m*)}
■

The above Subtype Lemma 5.61 says that verified superclass code will continue to satisfy its superclass specification, in the context of a new subclass, if the precondition of the superclass method and the subclass invariant is established prior to a downcall. That is, the expected postcondition, used in the superclass verification, requires that the subclass invariant be established prior to a downcall. However, since our technique does not allow superclass methods to invalidate the subclass parts of the run-time type invariant, establishing the invariant of the static type of the receiver automatically establishes the run-time type invariant (Valid Invariant Theorem 5.30), i.e., the subclass invariant. The next few lemmas use this property to prove that the superclass methods our rules allow to be called will continue to satisfy their superclass specification, i.e., they do not have to be re-verified.

Lemma 5.62 (Receiver Substitution): Let S1, S2 ∈ *State*. Let *P* be an expression that does not access local, stack variables other than the receiver `this`.

If *heapOf*(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r ∧ r ∈ *ObjectRef*
then S1<*P*[`this`←*vr1*]> = S2<*P*[`this`←*vr2*]>.
Proof:
The proof will be by induction on the structure of expression *P*.
Basis:
Case 1: *P* ≡ `this`
heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r
⇒ < both are equal to the same reference (r) >
S1<*vr1*> = S2<*vr2*>
⇒ < definition of substitution and *P* ≡ `this` >
S1<*P*[`this`←*vr1*]> = S2<*P*[`this`←*vr2*]>
Case 2: *P* ≡ `this.`*f*
heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r
∧ S1<*P*[`this`←*vr1*]> ∧ S2<*P*[`this`←*vr2*]>
⇔ < definition of substitution and *P* ≡ `this.`*f* >
heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r
∧ S1<*P*[`this`←*vr1*]> = S1<*vr1*`.`*f*> ∧ S2<*P*[`this`←*vr2*]> = S2<*vr2*`.`*f*>
⇔ < by the L-Field and E-VarRef rules of Figure 4.16 >

heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r

∧ S1<*P*[`this`←*vr1*]> = *getValue*(S1, *loc*(*f*, r))

∧ S2<*P*[`this`←*vr2*]> = *getValue*(S2, *loc*(*f*, r))

⇔ < by the definition of *getValue* of Figure 4.16 and because r ≠ `local`
(i.e., *vr1* **.** *f* and *vr2* **.** *f* are not local/stack variables) >

heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r

∧ S1<*P*[`this`←*vr1*]> = *heapOf*(S1) (*loc*(*f*, r))

∧ S2<*P*[`this`←*vr2*]> = *heapOf*(S2) (*loc*(*f*, r))

⇒ < since *heapOf*(S1) = *heapOf*(S2) >

S1<*P*[`this`←*vr1*]> = S2<*P*[`this`←*vr2*]>

Case 3: *P* ≡ `x`

Not allowed since *P* does not access local variables.

Case 4: *P* is a literal

heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r

∧ S1<*P*[`this`←*vr1*]> ∧ S2<*P*[`this`←*vr2*]>

⇒ < definition of substitution for *P* ≡ *lit* >

S1<*P*[`this`←*vr1*]> = S1<*lit*> ∧ S2<*P*[`this`←*vr2*]> = S2<*lit*>

⇔ < by the E-Literal rule of Figure 4.16 >

S1<*P*[`this`←*vr1*]> = *mkVal*(*lit*) ∧ S2<*P*[`this`←*vr2*]> = *mkVal*(*lit*)

⇔ < since both have the same value >

S1<*P*[`this`←*vr1*]> = S2<*P*[`this`←*vr2*]>

Case 5: *P* ≡ `( T ) null`

heapOf(S1) = *heapOf*(S2) ∧ S1<*vr1*> = r ∧ S2<*vr2*> = r

∧ S1<*P*[`this`←*vr1*]> ∧ S2<*P*[`this`←*vr2*]>

⇒ < definition of substitution for *P* ≡ `( T ) null` >

S1<*P*[`this`←*vr1*]> = S1<`( T ) null`> ∧ S2<*P*[`this`←*vr2*]> = S2<`( T ) null`>

⇒ < E-CastNull rule of Figure 4.16 >

S1<*P*[`this`←*vr1*]> = *voidV*(null) ∧ S2<*P*[`this`←*vr2*]> = *voidV*(null)

⇒ < since both have the same value >

S1<*P*[`this`←*vr1*]> = S2<*P*[`this`←*vr2*]>

Induction Step:

The induction hypothesis assumes that this Lemma holds for all subexpressions of a larger expression. We will start our calculations from this hypothesis.

Case 1: *P* ≡ `(` *e1* `)`

S1<*e1*[`this`←*vr1*]> = S2<*e1*[`this`←*vr2*]>

⇒ < the E-Paren rule of Figure 4.16 >

S1<`(` *e1* `)[this←`*vr1*`]` > = S2<`(` *e1* `)[this←`*vr2*`]` >

⇒ < definition of substitution and *P* ≡ (*e1*) >

S1<*P*`[this←`*vr1*`]` > = S2<*P*`[this←`*vr2*`]` >

Case 2: *P* ≡ *e1 bop e2*

S1<*e1*`[this←`*vr1*`]` > = S2<*e1*`[this←`*vr2*`]` >

∧ S1<*e2*`[this←`*vr1*`]` > = S2<*e2*`[this←`*vr2*`]` >

⇒ < E-BinOp rule of Figure 4.16 >

S1<*e1*`[this←`*vr1*`]` > = S2<*e1*`[this←`*vr2*`]` > = v1

∧ S1<*e2*`[this←`*vr1*`]` > = S2<*e2*`[this←`*vr2*`]` > = v2

∧ S1<*e1*`[this←`*vr1*`]` *bop e2*`[this←`*vr1*`]` > = *apply*(*bop*, v1, v2))

∧ S2<*e1*`[this←`*vr2*`]` *bop e2*`[this←`*vr2*`]` > = *apply*(*bop*, v1, v2))

⇒ < definition of substitution, *P* ≡ *e1 bop e2*, and both have the same value >

S1<*P*`[this←`*vr1*`]` > = S2<*P*`[this←`*vr2*`]` >

Case 3: *P* ≡ *uop e1*

S1<*e1*`[this←`*vr1*`]` > = S2<*e1*`[this←`*vr2*`]` >

⇒ < the E-UnOp rule of Figure 4.16 >

S1<*e1*`[this←`*vr1*`]` > = S2<*e1*`[this←`*vr2*`]` > = v

∧ S1<*uop e1*`[this←`*vr1*`]` > = *apply*(*uop*, v) ∧ S2<*uop e1*`[this←`*vr2*`]` > = *apply*(*uop*, v)

⇒ < definition of substitution, *P* ≡ *uop e1*, and both have the same value >

S1<*P*`[this←`*vr1*`]` > = S2<*P*`[this←`*vr2*`]` >

Case 4: *P* ≡ `(T)` *e1*

S1<*e1*`[this←`*vr1*`]` > = S2<*e1*`[this←`*vr2*`]` >

⇒ < the E-Cast rule of Figure 4.16 >

S1<*e1*`[this←`*vr1*`]` > = S2<*e1*`[this←`*vr2*`]` > = v

∧ S1<`(T)` *e1*`[this←`*vr1*`]` > = v ∧ S2<`(T)` *e1*`[this←`*vr2*`]` > = v

⇒ < definition of substitution, *P* ≡ `(T)`*e1*, and both have the same value >

S1<*P*`[this←`*vr1*`]` > = S2<*P*`[this←`*vr2*`]` >

■

In the next few Lemmas, 5.63-5.67, there are sometimes five or six lines of assertions because they include the operational semantics of the method call. Therefore, to make it easier to follow the proof, we have tried, whenever possible, to place the assertions that will be used in the next step of the calculation on the last one or two lines. Also, in most cases, we add any new assertions on or near the last line.

Lemma 5.63 (Self-Call): Let `this.`*m*(*e*) be a self-call allowed by our technique in some method U`.`*n*. Let S ∈ *State* be the pre-state prior to the execution of `this.`*m*(*e*) and let S1 ∈ *State* be the post-state after the execution of `this.`*m*(*e*). Let D be the run-time type of the receiver `this` and let *inv*(D) hold in the pre-state before the execution of U`.`*n*.

If T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)},
then S1<*inv*(D) && *ens*(T, *m*)>.
Proof:

U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}
⇒ < D is the run-time type of the receiver and U is the static type of the receiver, so D ≤ U;
thus, D ≤ T2 ≤ T, since T = *whereMethodDecl*(U, *m*) and T2 = *whereMethodDecl*(D, *m*) >
U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}
⇒ < by the Subtype Lemma 5.61 >
U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
⇒ < the precondition as given in the A-SelfCall rule of Figure 4.21 must hold
in the pre-state since the call is not allowed otherwise >
U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*inv*(U) && *req*(T, *m*)[`p`←*e*]>
⇒ < semantics of the && operator >
U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[`p`←*e*]> ∧ S<*inv*(U)>
⇒ < by the Valid Invariant Theorem 5.30, the run-time type invariant, *inv*(D), is established
because *inv*(U), the invariant of the static type of the receiver, is established
and *inv*(D) held in the pre-state prior to the execution of U`.`*n* >
U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[`p`←*e*]> ∧ S<*inv*(D)>
⇒ < by the operational semantics of the S-Call rule of Figure 4.16 and
since D is the run-time type of the receiver >
U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ D = *refType*(r) ∧ [`this`, S] ⇒$_e$ r ∧ [*e*, S] ⇒$_e$ v
∧ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc :=` r][*loc*(`p`, `local`) `:=` v]
∧ {*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {`this.`*m*`(`*e*`)`, S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[`p`←*e*]> ∧ S<*inv*(D)>
⇒ < by the Substitution Theorem 5.9 and because *req*(T, *m*) is a function of the receiver

and formal parameter and *inv*(D) is a function of the receiver >

U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ D = *refType*(r) ∧ [`this`, S] ⇒$_e$ r ∧ [*e*, S] ⇒$_e$ v
∧ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc :=` r][*loc*(`p`, `local`) `:=` v]
∧ {*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {`this.`*m*`(`*e*`)`, S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[`p`←*e*]> ∧ S<*inv*(D)>
∧ S<*req*(T, *m*)[`p`←*e*]> = S′<*req*(T, *m*)> ∧ S<*inv*(D)> = S′<*inv*(D)>

⇒ < equality and the Invariant Clause Lemma 5.1 >

U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*) ∧ T2 = *whereMethDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ D = *refType*(r) ∧ [`this`, S] ⇒$_e$ r ∧ [*e*, S] ⇒$_e$ v
∧ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc :=` r][*loc*(`p`, `local`) `:=` v]
∧ {*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {`this.`*m*`(`*e*`)`, S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S′<*req*(T, *m*)> ∧ S′<*inv*(T2)> ∧ S′<*inv*(D)>

⇒ < by the meaning of |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)} >

{*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {`this.`*m*`(`*e*`)`, S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ S″<*inv*(T2) && *ens*(T, *m*)> ∧ S′<*inv*(D)>

⇒ < by the Valid Invariant Theorem 5.30, the run-time type invariant, *inv*(D), is established since a method in T2 established *inv*(T2) and the run-time invariant held in the pre-state S′ >

{*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {`this.`*m*`(`*e*`)`, S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ S″<*inv*(D) && *ens*(T, *m*)>

⇒ < since S1 is the post-state after the execution of `this.`*m*`(`*e*`)` >

{*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {`this.`*m*`(`*e*`)`, S} ⇒$_s$ S1
∧ S″<*inv*(D) && *ens*(T, *m*)> ∧ S1 = *state*(*stackOf*(S), *heapOf*(S″))

⇒ < the predicate *inv*(D) && *ens*(T, *m*) is not allowed to access local variables on the post-state stack of S″, the receiver object is the same in both S1 and S″, and both S1 and S″ have the same heap; thus by the Receiver Substitution Lemma 5.62 and since *P* = *P*[`this`←`this`] >

S″<*inv*(D) && *ens*(T, *m*)> ∧ S″<*inv*(D) && *ens*(T, *m*)> = S1<*inv*(D) && *ens*(T, *m*)>

⇒ < equality >

S1<*inv*(D) && *ens*(T, *m*)>

■

Lemma 5.64 (Super-Call): Let `super.`*m*(*e*) be a super-call allowed by our technique in some method U`.`*n*. Let S ∈ *State* be the pre-state prior to the execution of `super.`*m*(*e*) and let S1 ∈ *State* be the

post-state after the execution of `super.`*m*(*e*). Let D be the run-time type of `this` and let *inv*(D) hold in the pre-state before the execution of U`.`*n*.

If T = *whereMethodDecl*(*superOf*(U), *m*)
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)},
then S1<*inv*(D) && *ens*(T, *m*)>.
Proof:

U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*)
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\Rightarrow$ < U is the static type of the receiver, so D $\leq$ U $\leq$ T >
U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\Rightarrow$ < the precondition as given in the A-SupCall rule of Figure 4.21 must hold in the pre-state since the call is not allowed otherwise >
U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\wedge$ S<*inv*(U) && *req*(T, *m*)[p$\leftarrow$*e*]>
$\Rightarrow$ < semantics of the && operator >
U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\wedge$ S<*req*(T, *m*)[p$\leftarrow$*e*]> $\wedge$ S<*inv*(U)>
$\Rightarrow$ < by the Valid Invariant Theorem 5.30, the run-time type invariant, *inv*(D), is established because *inv*(U), the invariant of the static type of the receiver, is established and *inv*(D) held in the pre-state prior to the execution of U`.`*n* >
U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\wedge$ S<*req*(T, *m*)[p$\leftarrow$*e*]> $\wedge$ S<*inv*(D)>
$\Rightarrow$ < by the operational semantics of the S-SupCall rule in Figure 4.18 >
U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ [`this`, S] $\Rightarrow_e$ r $\wedge$ [*e*, S] $\Rightarrow_e$ v
$\wedge$ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc := `r][*loc*(p, `local`) `:=` v]
$\wedge$ {*getBody*(T, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {`super.`*m*`(`*e*`)`, S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ $\models$ {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\wedge$ S<*req*(T, *m*)[p$\leftarrow$*e*]> $\wedge$ S<*inv*(D)>
$\Rightarrow$ < by the Substitution Theorem 5.9 and because *req*(T, *m*) is a function of the receiver and formal parameter and *inv*(D) is a function of the receiver >
U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ [`this`, S] $\Rightarrow_e$ r $\wedge$ [*e*, S] $\Rightarrow_e$ v

$\wedge$ S′ = *state*(emptyS, *heapOf*(S))[thisloc := r][*loc*(p, local) := v]
$\wedge$ {*getBody*(T, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {super.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ |= {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\wedge$ S<*req*(T, *m*)[p←*e*]> $\wedge$ S<*inv*(D)>
$\wedge$ S<*req*(T, *m*)[p←*e*]> = S′<*req*(T, *m*)> $\wedge$ S<*inv*(D)> = S′<*inv*(D)>

$\Rightarrow$ < equality and the Invariant Clause Lemma 5.1 >
U = *typeOf*(this) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*) $\wedge$ D $\leq$ U $\leq$ T
$\wedge$ [this, S] $\Rightarrow_e$ r $\wedge$ [*e*, S] $\Rightarrow_e$ v
$\wedge$ S′ = *state*(emptyS, *heapOf*(S))[thisloc := r][*loc*(p, local) := v]
$\wedge$ {*getBody*(T, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {super.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ |= {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)}
$\wedge$ S′<*req*(T, *m*)> $\wedge$ S′<*inv*(T)> $\wedge$ S′<*inv*(D)>

$\Rightarrow$ < by the meaning of |= {*inv*(T) && *req*(T, *m*)} *getBody*(T, *m*) {*inv*(T) && *ens*(T, *m*)} >
{*getBody*(T, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {super.*m*(*e*), S′} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ S″<*inv*(T) && *ens*(T, *m*)> $\wedge$ S′<*inv*(D)>

$\Rightarrow$ < by the Valid Invariant Theorem 5.30, the run-time type invariant is established
since a method in T established *inv*(T) and *inv*(D) held in the pre-state S′ >
{*getBody*(T, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {super.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ S″<*inv*(D) && *ens*(T, *m*)>

$\Rightarrow$ < since S1 is the post-state after the execution of super.*m*(*e*) >
{*getBody*(T, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {super.*m*(*e*), S} $\Rightarrow_s$ S1
$\wedge$ S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S″))

$\Rightarrow$ < the predicate *inv*(D) && *ens*(T, *m*) is not allowed to access local variables on
the post-state stack of S″, the receiver object is the same in both S1 and S″, and
both S1 and S″ have the same heap; thus by the Receiver Substitution Lemma 5.62
and since [this←this] does not change the assertion >
S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ S″<*inv*(D) && *ens*(T, *m*)> = S1<*inv*(D) && *ens*(T, *m*)>

$\Rightarrow$ < equality >
S1<*inv*(D) && *ens*(T, *m*)>

■

Lemma 5.65 (Object-Call): Let *rcvr*.*m*(*e*) be an object-call allowed by our technique in some method U.*n*. Let S $\in$ *State* be the pre-state prior to the execution of *rcvr*.*m*(*e*) and let S1 $\in$ *State* be the post-state after the execution of *rcvr*.*m*(*e*). Let D be the run-time type of *rcvr*.

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)
$\wedge$ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)},
$\wedge$ S<*inv*(D)[this←*rcvr*]>
then S1<*inv*(D)[this←*rcvr*] && *ens*(T, *m*)[this←*rcvr*]>.

Proof:

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*)
∧ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}
∧ S<*inv*(D)[this←*rcvr*]>
⇒ < T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) and T2 = *whereMethodDecl*(D, *m*)
and D is the run-time type of *rcvr*, so D ≤ T2 ≤ T >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*) ∧ D ≤ T2 ≤ T
∧ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}
∧ S<*inv*(D)[this←*rcvr*]>
⇒ < by the Subtype Lemma 5.61 >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*) ∧ D ≤ T2 ≤ T
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*inv*(D)[this←*rcvr*]>
⇒ < the precondition as given in the A-ObjCall rule of Figure 4.21 must hold
in the pre-state since the call is not allowed otherwise >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*) ∧ D ≤ T2 ≤ T
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[this←*rcvr*, p←*e*]> ∧ S<*inv*(D)[this←*rcvr*]>
⇒ < by the operational semantics of an object-call given in the S-Call rule of Figure 4.18 >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ [*rcvr*, S] ⇒$_e$ r ∧ [*e*, S] ⇒$_e$ v ∧ D = *refType*(r)
∧ S′ = *state*(emptyS, *heapOf*(S))[thisloc := r][*loc*(p, local) := v]
∧ {*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {*rcvr*.*m*(*e*), S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[this←*rcvr*, p←*e*]> ∧ S<*inv*(D)[this←*rcvr*]>
⇒ < by the Substitution Theorem 5.9 and because *req*(T, *m*) is a function of the receiver
and formal parameter and *inv*(D) is a function of the receiver >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ [*rcvr*, S] ⇒$_e$ r ∧ [*e*, S] ⇒$_e$ v ∧ D = *refType*(r)
∧ S′ = *state*(emptyS, *heapOf*(S))[thisloc := r][*loc*(p, local) := v]
∧ {*getBody*(T2, *m*), S′} ⇒$_s$ S″ ∧ {*rcvr*.*m*(*e*), S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}
∧ S<*req*(T, *m*)[this←*rcvr*, p←*e*]> ∧ S′<*req*(T, *m*)> = S<*req*(T, *m*)[this←*rcvr*, p←*e*]>
∧ S<*inv*(D)[this←*rcvr*]> ∧ S′<*inv*(D)> = S<*inv*(D)[this←*rcvr*]>
⇒ < by equality and the Invariant Clause Lemma 5.1 >
T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) ∧ T2 = *whereMethodDecl*(D, *m*)
∧ D ≤ T2 ≤ T ∧ [*rcvr*, S] ⇒$_e$ r ∧ [*e*, S] ⇒$_e$ v ∧ D = *refType*(r)

$\wedge$ S′ = *state*(emptyS, *heapOf*(S))[thisloc := r][*loc*(p, local) := v]
$\wedge$ {*getBody*(T2, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {*rcvr*.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ S′<*req*(T, *m*)> $\wedge$ S′<*inv*(T2)> $\wedge$ S′<*inv*(D)>
$\wedge$ |= {*inv*(T2) && *req*(T, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T, *m*)}

$\Rightarrow$ < by the meaning of the last conjunct and the operational semantics of the call >
S′ = *state*(emptyS, *heapOf*(S))[thisloc := r][*loc*(p, local) := v]
$\wedge$ {*getBody*(T2, *m*), S′} $\Rightarrow_s$ S″ $\wedge$ {*rcvr*.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ [*rcvr*, S] $\Rightarrow_e$ r $\wedge$ S″<*inv*(T2) && *ens*(T, *m*)> $\wedge$ S′<*inv*(D)>

$\Rightarrow$ < the same two parameters (this and p) will be on the stack at the end of the execution of T2.*m*, i.e., in both S′ and S″ >
{*rcvr*.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ [*rcvr*, S] $\Rightarrow_e$ r $\wedge$ S″<*inv*(T2) && *ens*(T, *m*)> $\wedge$ S′<*inv*(D)>
$\wedge$ [this, S″] $\Rightarrow_e$ r

$\Rightarrow$ < by the Valid Invariant Theorem 5.30, the run-time type invariant is established since a method in T2 established *inv*(T2) and *inv*(D) held in the pre-state S′ >
{*rcvr*.*m*(*e*), S} $\Rightarrow_s$ *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ [*rcvr*, S] $\Rightarrow_e$ r $\wedge$ S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ [this, S″] $\Rightarrow_e$ r

$\Rightarrow$ < since S1 is the post-state after the execution of *rcvr*.*m*(*e*) >
{*rcvr*.*m*(*e*), S} $\Rightarrow_s$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ [*rcvr*, S] $\Rightarrow_e$ r $\wedge$ S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ [this, S″] $\Rightarrow_e$ r

$\Rightarrow$ < by the E-VarRef rule of Figure 4.16 >
{*rcvr*.*m*(*e*), S} $\Rightarrow_s$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ [this, S″] $\Rightarrow_e$ r
$\wedge$ [*rcvr*, S] $\Rightarrow_{lv}$ vLoc $\wedge$ [*rcvr*, S] $\Rightarrow_e$ *getValue*(S, vLoc) $\wedge$ r = *getValue*(S, vLoc)

$\Rightarrow$ < since S1 has the same stack as S, and *rcvr* has to be a local variable or a pivot field of the receiver this, and *heapOf*(S″) is an extension of *heapOf*(S) >
{*rcvr*.*m*(*e*), S} $\Rightarrow_s$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ [this, S″] $\Rightarrow_e$ r
$\wedge$ [*rcvr*, S1] $\Rightarrow_{lv}$ vLoc $\wedge$ [*rcvr*, S] $\Rightarrow_e$ *getValue*(S, vLoc) $\wedge$ r = *getValue*(S, vLoc)

$\Rightarrow$ < location vLoc cannot be assigned during the call of *rcvr*.*m*(*e*) if *rcvr* is a local variable of the calling method U.*n*; also, location vLoc cannot be assigned if *rcvr* is a pivot field because a pivot field cannot be the receiver when the called method has side-effects and its formal parameter references the receiver of the calling method U.*n* (by the Actual Parameter Aliasing Lemma 5.24) >
{*rcvr*.*m*(*e*), S} $\Rightarrow_s$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S″))
$\wedge$ S″<*inv*(D) && *ens*(T, *m*)> $\wedge$ [this, S″] $\Rightarrow_e$ r
$\wedge$ [*rcvr*, S1] $\Rightarrow_{lv}$ vLoc $\wedge$ [*rcvr*, S1] $\Rightarrow_e$ *getValue*(S, vLoc) $\wedge$ r = *getValue*(S1, vLoc)

$\Rightarrow <$ by the E-VarRef rule of Figure 4.16 $>$

$\{rcvr\,.\,m\,(e)\,, \mathsf{S}\} \Rightarrow_s \mathsf{S1} \wedge \mathsf{S1} = state(stackOf(\mathsf{S}), heapOf(\mathsf{S}''))$

$\wedge\ \mathsf{S}''<inv(\mathrm{D})\ \&\&\ ens(\mathrm{T}, m)> \wedge [\texttt{this}, \mathsf{S}''] \Rightarrow_e \mathrm{r} \wedge [rcvr, \mathsf{S1}] \Rightarrow_e \mathrm{r}$

$\Rightarrow <$ the predicate $inv(\mathrm{D})\ \&\&\ ens(\mathrm{T}, m)$ is not allowed to access local variables on the post-state stack of S'', $rcvr$ in $\mathsf{S1}$ and `this` in S'' reference the same object, and both $\mathsf{S1}$ and S'' have the same heap; thus by the Receiver Substitution Lemma 5.62 and since $P = P[\texttt{this}\leftarrow\texttt{this}]$ $>$

$\mathsf{S}''<inv(\mathrm{D})\ \&\&\ ens(\mathrm{T}, m)>$

$\wedge\ \mathsf{S}''<inv(\mathrm{D})\ \&\&\ ens(\mathrm{T}, m)> = \mathsf{S1}<inv(\mathrm{D})[\texttt{this}\leftarrow rcvr]\ \&\&\ ens(\mathrm{T}, m)[\texttt{this}\leftarrow rcvr]>$

$\Rightarrow <$ equality $>$

$\mathsf{S1}<inv(\mathrm{D})[\texttt{this}\leftarrow rcvr]\ \&\&\ ens(\mathrm{T}, m)[\texttt{this}\leftarrow rcvr]>$

■

Lemma 5.66 (Constructor Call): Let $vr = \texttt{new}\ \mathrm{T}(e)$ be a new object constructor call allowed by our technique in some method $\mathrm{U}\,.\,n$. Let $\mathsf{S} \in State$ be the pre-state prior to the call of $vr = \texttt{new}\ \mathrm{T}(e)$ and let $\mathsf{S1} \in State$ be the post-state after the execution of $vr = \texttt{new}\ \mathrm{T}(e)$.

If $\models \{req(\mathrm{T}, \mathrm{T})\}\ getBody(\mathrm{T}, \mathrm{T})\ \{inv(\mathrm{T})\ \&\&\ ens(\mathrm{T}, \mathrm{T})\}$,

then $\mathsf{S1}<inv(\mathrm{T})[\texttt{this}\leftarrow vr]\ \&\&\ ens(\mathrm{T}, \mathrm{T})[\texttt{this}\leftarrow vr]>$.

Proof:

$\models \{req(\mathrm{T}, \mathrm{T})\}\ getBody(\mathrm{T}, \mathrm{T})\ \{inv(\mathrm{T})\ \&\&\ ens(\mathrm{T}, \mathrm{T})\}$

$\Rightarrow <$ the precondition as given in the A-NewAssign rule of Figure 4.22 must hold in the pre-state since the call is not allowed otherwise $>$

$\models \{req(\mathrm{T}, \mathrm{T})\}\ getBody(\mathrm{T}, \mathrm{T})\ \{inv(\mathrm{T})\ \&\&\ ens(\mathrm{T}, \mathrm{T})\}$

$\wedge\ \mathsf{S}<req(\mathrm{T}, \mathrm{T})[\mathrm{p}\leftarrow e]>$

$\Rightarrow <$ by the operational semantics of the S-NewAssign rule in Figure 4.19 $>$

$\models \{req(\mathrm{T}, \mathrm{T})\}\ getBody(\mathrm{T}, \mathrm{T})\ \{inv(\mathrm{T})\ \&\&\ ens(\mathrm{T}, \mathrm{T})\}$

$\wedge\ [vr, \mathsf{S}] \Rightarrow_{lv} \mathrm{varLoc} \wedge [e, \mathsf{S}] \Rightarrow_e \mathrm{v} \wedge new(\mathsf{S}, \mathrm{T}) = (\mathrm{r}, \mathsf{S}')$

$\wedge\ \mathsf{S}'' = state(\texttt{emptyS}, heapOf(\mathsf{S}'))[\texttt{thisloc} := \mathrm{r}][loc(\mathrm{p}, \texttt{local}) := \mathrm{v}]$

$\wedge\ \{getBody(\mathrm{T}, m), \mathsf{S}''\} \Rightarrow_s \mathsf{S}'''$

$\wedge\ \{vr{=}\texttt{new}\ \mathrm{T}(e), \mathsf{S}\} \Rightarrow_s state(stackOf(\mathsf{S}), heapOf(\mathsf{S}'''))[\mathrm{varLoc} := \mathrm{r}]$

$\wedge\ \mathsf{S}<req(\mathrm{T}, \mathrm{T})[\mathrm{p}\leftarrow e]>$

$\Rightarrow <$ by the Substitution Theorem 5.9 and because the precondition cannot reference the state of the new object (which has not yet been initialized) $>$

$\models \{req(\mathrm{T}, \mathrm{T})\}\ getBody(\mathrm{T}, \mathrm{T})\ \{inv(\mathrm{T})\ \&\&\ ens(\mathrm{T}, \mathrm{T})\}$

$\wedge\ [vr, \mathsf{S}] \Rightarrow_{lv} \mathrm{varLoc} \wedge [e, \mathsf{S}] \Rightarrow_e \mathrm{v} \wedge new(\mathsf{S}, \mathrm{T}) = (\mathrm{r}, \mathsf{S}')$

$\wedge\ \mathsf{S}'' = state(\texttt{emptyS}, heapOf(\mathsf{S}'))[\texttt{thisloc} := \mathrm{r}][loc(\mathrm{p}, \texttt{local}) := \mathrm{v}]$

$\wedge\ \{getBody(\mathrm{T}, m), \mathsf{S}''\} \Rightarrow_s \mathsf{S}'''$

$\wedge\ \{vr{=}\texttt{new}\ \mathrm{T}(e), \mathsf{S}\} \Rightarrow_s state(stackOf(\mathsf{S}), heapOf(\mathsf{S}'''))[\mathrm{varLoc} := \mathrm{r}]$

$\wedge$ S<*req*(T, T)[p←*e*]> $\wedge$ S<*req*(T, T)[p←*e*]> = S′<*req*(T, T)>

$\Rightarrow$ < equality >

|= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

$\wedge$ [*vr*, S] $\Rightarrow_{lv}$ varLoc $\wedge$ [*e*, S] $\Rightarrow_{e}$ v $\wedge$ *new*(S, T) = (r, S′)

$\wedge$ S″ = *state*(emptyS, *heapOf*(S′))[thisloc := r][*loc*(p, local) := v]

$\wedge$ {*getBody*(T, *m*), S″} $\Rightarrow_{s}$ S‴

$\wedge$ {*vr*=new T(*e*), S} $\Rightarrow_{s}$ *state*(*stackOf*(S), *heapOf*(S‴))[varLoc := r]

$\wedge$ S′<*req*(T, T)>

$\Rightarrow$ < by the meaning of |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)} on line 1 and the operational semantics of the call >

S″ = *state*(emptyS, *heapOf*(S′))[thisloc := r][*loc*(p, local) := v]

$\wedge$ {*getBody*(T, *m*), S″} $\Rightarrow_{s}$ S‴

$\wedge$ {*vr*=new T(*e*), S} $\Rightarrow_{s}$ *state*(*stackOf*(S), *heapOf*(S‴))[varLoc := r]

$\wedge$ S‴<*inv*(T) && *ens*(T, T)> $\wedge$ [*vr*, S] $\Rightarrow_{lv}$ varLoc

$\Rightarrow$ < since this in S″ and S‴ reference the same object r (i.e., methods cannot assign to this) >

{*vr*=new T(*e*), S} $\Rightarrow_{s}$ *state*(*stackOf*(S), *heapOf*(S‴))[varLoc := r]

$\wedge$ S‴<*inv*(T) && *ens*(T, T)> $\wedge$ [this, S‴] $\Rightarrow_{e}$ r $\wedge$ [*vr*, S] $\Rightarrow_{lv}$ varLoc

$\Rightarrow$ < since S1 is the post-state after the execution of *vr*=new T(*e*) >

{*vr*=new T(*e*), S} $\Rightarrow_{s}$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S‴))[varLoc := r]

$\wedge$ S‴<*inv*(T) && *ens*(T, T)> $\wedge$ [this, S‴] $\Rightarrow_{e}$ r $\wedge$ [*vr*, S] $\Rightarrow_{lv}$ varLoc

$\Rightarrow$ < since S1 has the same stack as S, and *vr* has to be a local variable or a field of the receiver this (syntax of assignment), and *heapOf*(S‴) is an extension of *heapOf*(S) >

{*vr*=new T(*e*), S} $\Rightarrow_{s}$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S‴))[varLoc := r]

$\wedge$ S‴<*inv*(T) && *ens*(T, T)> $\wedge$ [this, S‴] $\Rightarrow_{e}$ r $\wedge$ [*vr*, S1] $\Rightarrow_{lv}$ varLoc

$\Rightarrow$ < by the State Update Lemma 5.6 (since the update is [varLoc := r]) >

S‴<*inv*(T) && *ens*(T, T)> $\wedge$ [*vr*, S1] $\Rightarrow_{e}$ r $\wedge$ [this, S‴] $\Rightarrow_{e}$ r

$\wedge$ {*vr*=new T(*e*), S} $\Rightarrow_{s}$ S1 $\wedge$ S1 = *state*(*stackOf*(S), *heapOf*(S‴))[varLoc := r]

$\Rightarrow$ < the predicate *inv*(T) && *ens*(T, T) is not allowed to access local variables on the post-state stack of S‴, *vr* in S1 and this in S‴ reference the same object r, and both S1 and S‴ have the same heap; thus by the Receiver Substitution Lemma 5.62 and since *P* = *P*[this←this] >

S‴<*inv*(T) && *ens*(T, T)>

$\wedge$ S‴<*inv*(T) && *ens*(T, T)> = S1<*inv*(T)[this←*vr*] && *ens*(T, T)[this←*vr*]>

$\Rightarrow$ < equality >

S1<*inv*(T)[this←*vr*] && *ens*(T, T)[this←*vr*]>

■

Lemma 5.67 (Superclass Constructor): Let `super`(*e*) be a superclass constructor call allowed by our technique in some constructor U . U. Let S ∈ *State* be the pre-state prior to the call of `super`(*e*) and let S1 ∈ *State* be the post-state after the execution of `super`(*e*).

If T = *superOf*(U) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)},

then S1<*inv*(T) && *ens*(T, T)>.

Proof:

T = *superOf*(U) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

⇒ < the precondition as given in the A-SupConstr rule of Figure 4.21 must hold in the pre-state since the call is not allowed otherwise >

T = *superOf*(U) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}
∧ S<*req*(T, T)[p←*e*]>

⇒ < by the operational semantics of the S-NewAssign rule in Figure 4.19 >

T = *superOf*(U) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}
∧ [*e*, S] ⇒$_e$ v ∧ [`this`, S] ⇒$_e$ r
∧ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc` := r][*loc*(`p`, `local`) := v]
∧ {*getBody*(T, *m*), S′} ⇒$_s$ S″ ∧ {`super`(*e*), S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ S<*req*(T, T)[p←*e*]>

⇒ < by the Substitution Theorem 5.9 and because the precondition cannot reference the state of the new object (which has not yet been initialized) >

T = *superOf*(U) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}
∧ [*e*, S] ⇒$_e$ v ∧ [`this`, S] ⇒$_e$ r
∧ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc` := r][*loc*(`p`, `local`) := v]
∧ {*getBody*(T, *m*), S′} ⇒$_s$ S″ ∧ {`super`(*e*), S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ S<*req*(T, T)[p←*e*]> ∧ S<*req*(T, T)[p←*e*]> = S′<*req*(T, T)>

⇒ < equality >

T = *superOf*(U) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}
∧ [*e*, S] ⇒$_e$ v ∧ [`this`, S] ⇒$_e$ r
∧ S′ = *state*(`emptyS`, *heapOf*(S))[`thisloc` := r][*loc*(`p`, `local`) := v]
∧ {*getBody*(T, *m*), S′} ⇒$_s$ S″ ∧ {`super`(*e*), S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))
∧ S′<*req*(T, T)>

⇒ < by the meaning of |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)} on line 1 and the operational semantics of the call >

S″<*inv*(T) && *ens*(T, T)> ∧ {`super`(*e*), S} ⇒$_s$ *state*(*stackOf*(S), *heapOf*(S″))

⇒ < since S1 is the post-state after the execution of `super`(*e*) >

S″<*inv*(T) && *ens*(T, T)> ∧ {`super`(*e*), S} ⇒$_s$ S1
∧ S1 = *state*(*stackOf*(S), *heapOf*(S″))

⇒ < the predicate *inv*(T) && *ens*(T, T) is not allowed to access local variables on

the post-state stack of S″, the receiver object is the same in both S1 and S″,
and both S1 and S″ have the same heap; thus by the Receiver Substitution Lemma 5.62
and since [this←this] does not change the assertion >

S″<*inv*(T) && *ens*(T, T)> ∧ S″<*inv*(T) && *ens*(T, T)> = S1<*inv*(T) && *ens*(T, T)>

⇒ < equality >

S1<*inv*(T) && *ens*(T, T)>

■

Recall from subsection 3.2.1 that an *access path* is a sequence of variable or field names with each name denoting a context (i.e., an object) in which the succeeding name is resolved. We now define a *pivot field access path* to be an access path that starts with an owner variable followed by a sequence of pivot field names. For example, `p.x.y` would be a pivot field access path if `p` is a formal parameter (an owner), `x` is a pivot field in `p`, and `y` is a pivot field in `p.x`.

Theorem 5.68 (Method Call): Let method U.*n* make a super-call, self-call, object-call, new object constructor call, or superclass constructor call. Let *rcvr* be the receiver, *m* be the method or constructor called, and *e* be the actual argument. Let S ∈ *State* be the pre-state prior to the execution of *m*(*e*) and let S′ ∈ *State* be the post-state after the execution of *m*(*e*). Let D be the run-time type of *rcvr* and let D1 be the run-time type of parameter *e*. Assume that the run-time type invariant holds, in the pre-state S, for all objects referenced by a pivot field access path that starts with an owner variable visible in *m*.

If T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*)

∧ (∀ T1 ∈ *TypeId*, *n* ∈ *MethId* :

{*inv*(T1) && *req*(T1, *n*)} *getBody*(T1, *n*) {*inv*(T1) && *ens*(T1, *n*)}),

∧ (∀ T1 ∈ *TypeId* :

{*req*(T1, T1)} *getBody*(T1, T1) {*inv*(T1) && *ens*(T1, T1)}),

then S′<*inv*(D)[this←*rcvr*] && *ens*(T, *m*)[this←*rcvr*]>

∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[this←*e*]>).

Proof:

Our assumption about objects referenced by a pivot field access path requires that the run-time type invariant of the receiver and formal parameter (if it references an object) hold in the pre-state since they are owners (see predicate *isOwner* of Figure 5.5). It also says that the pivot fields of the receiver must contain `null` or a reference to an object whose run-time type invariant holds in the pre-state. Similarly, all objects reachable through pivot field access paths must have an invariant that holds in the pre-state.

If the call *m*(*e*) terminates, then there must be a maximum length of the call chains during the execution of T.*m*. Therefore, the proof will be by induction on the maximum length of the valid call chains during the call of *m*(*e*).

Basis: *k* = 0, i.e., *m*(*e*) makes no calls.

Case 1: $m(e)$ is an object-call, e.g., $rcvr\,.\,m(e)$

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\Rightarrow$ < from our assumption, the run-time type invariant of *rcvr* and *e* (if it references an object) must hold in the pre-state since they have to be owners (by the T-Call rule of Figure 5.1 and predicate *invariantOk* of Figure 5.4) >

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\wedge$ S<*inv*(D)[`this`$\leftarrow$*rcvr*]> $\wedge$ (D1 $\notin$ *TypeId* $\vee$ S<*e* == null> $\vee$ S<*inv*(D1)[`this`$\leftarrow$*e*]>)

$\Rightarrow$ < since there are no calls during the execution of $rcvr\,.\,m(e)$, the state of object *e* cannot be changed, so the invariant of *e* (if it references an object) must hold in the post-state S′ >

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\wedge$ S<*inv*(D)[`this`$\leftarrow$*rcvr*]> $\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<*e* == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow$*e*]>)

$\Rightarrow$ < since there are no calls during the execution of $rcvr\,.\,m(e)$, the correctness proof of method T2`.`*m* must be valid since all of the statements executed preserve validity (Lemmas 5.49 - 5.60) >

T = *whereMethodDecl*(*typeOf*(*rcvr*), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ |= {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\wedge$ S<*inv*(D)[`this`$\leftarrow$*rcvr*]> $\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<*e* == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow$*e*]>)

$\Rightarrow$ < by the Object-Call Lemma 5.65 >

S′<*inv*(D)[`this`$\leftarrow$*rcvr*] && *ens*(T, *m*)[`this`$\leftarrow$*rcvr*]>

$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<*e* == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow$*e*]>)

Case 2: $m(e)$ is a self-call, e.g., `this`$\,.\,m(e)$

T = *whereMethodDecl*(*typeOf*(`this`), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\Rightarrow$ < from our assumption, the run-time type invariant of *e* (if it references an object) must hold in the pre-state since it has to be an owner (by the T-Call rule of Figure 5.1 and predicate *invariantOk* of Figure 5.4) >

T = *whereMethodDecl*(*typeOf*(`this`), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S<*e* == null> $\vee$ S<*inv*(D1)[`this`$\leftarrow$*e*]>)

$\Rightarrow$ < since there are no calls during the execution of `this`$\,.\,m(e)$, the state of object *e* cannot be changed, so the invariant of object *e* must hold in the post-state S′ >

T = *whereMethodDecl*(*typeOf*(`this`), *m*) $\wedge$ T2 = *whereMethodDecl*(D, *m*)

$\wedge$ {*inv*(T2) && *req*(T2, *m*)} *getBody*(T2, *m*) {*inv*(T2) && *ens*(T2, *m*)}

$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<*e* == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow$*e*]>)

$\Rightarrow <$ since there are no calls during the execution of `this.`$m(e)$, the correctness proof of method T2`.`m must be valid since all of the statements executed preserve validity (Lemmas 5.49 - 5.60) $>$

T = *whereMethodDecl*(*typeOf*(`this`), m) $\wedge$ T2 = *whereMethodDecl*(D, m)
$\wedge$ $\models$ {*inv*(T2) && *req*(T2, m)} *getBody*(T2, m) {*inv*(T2) && *ens*(T2, m)}
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<e == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow e$]>)

$\Rightarrow <$ by the Self-Call Lemma 5.63 $>$

S′<*inv*(D) && *ens*(T, m)>
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<e == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow e$]>)

$\Rightarrow <$ substitution of `this` for `this` does not change the value of the expression $>$

S′<*inv*(D)[`this`$\leftarrow$`this`] && *ens*(T, m)[`this`$\leftarrow$`this`]>
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<e == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow e$]>)

Case 3: $m(e)$ is a super-call, e.g., `super.`$m(e)$

T = *whereMethodDecl*(*typeOf*(`super`), m) $\wedge$ *typeOf*(`super`) = *superOf*(*typeOf*(`this`))
$\wedge$ {*inv*(T) && *req*(T, m)} *getBody*(T, m) {*inv*(T) && *ens*(T, m)}

$\Rightarrow <$ from our assumption, the run-time type invariant of e (if it references an object) must hold in the pre-state since it has to be an owner (by the T-Call rule of Figure 5.1 and predicate *invariantOk* of Figure 5.4) $>$

T = *whereMethodDecl*(*typeOf*(`super`), m) $\wedge$ *typeOf*(`super`) = *superOf*(*typeOf*(`this`))
$\wedge$ {*inv*(T) && *req*(T, m)} *getBody*(T, m) {*inv*(T) && *ens*(T, m)}
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S<e == null> $\vee$ S<*inv*(D1)[`this`$\leftarrow e$]>)

$\Rightarrow <$ since there are no calls during the execution of `super.`$m(e)$, the state of object e cannot be changed, so the invariant of object e must hold in the post-state S′ $>$

T = *whereMethodDecl*(*typeOf*(`this`), m) $\wedge$ T2 = *whereMethodDecl*(D, m)
$\wedge$ {*inv*(T) && *req*(T, m)} *getBody*(T, m) {*inv*(T) && *ens*(T, m)}
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<e == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow e$]>)

$\Rightarrow <$ since there are no calls during the execution of `super.`$m(e)$, the correctness proof of method T2`.`m must be valid since all of the statements executed preserve validity (Lemmas 5.49 - 5.60) $>$

T = *whereMethodDecl*(*typeOf*(`this`), m) $\wedge$ T2 = *whereMethodDecl*(D, m)
$\wedge$ $\models$ {*inv*(T) && *req*(T, m)} *getBody*(T, m) {*inv*(T) && *ens*(T, m)}
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<e == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow e$]>)

$\Rightarrow <$ by the Super-Call Lemma 5.64 $>$

S′<*inv*(D) && *ens*(T, m)>
$\wedge$ (D1 $\notin$ *TypeId* $\vee$ S′<e == null> $\vee$ S′<*inv*(D1)[`this`$\leftarrow e$]>)

$\Rightarrow <$ `this` is the receiver in a super-call so we substitute `this` for `this` which does not change the value of the expression $>$

S′<*inv*(D)[this←this] && *ens*(T, *m*)[this←this]>

∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[this←*e*]>)

Case 4: *m*(*e*) is a new object constructor call, e.g., *rcvr*=new T(*e*)

Note that in this case, the target variable of the assignment becomes the receiver in the post-state after the call. Therefore, to match the theorem, we consider this target variable to be the receiver object *rcvr*.

T = *typeOf*(new T(*e*)) ∧ {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

⇒ < from our assumption, the run-time type invariant of *e* (if it references an object) must hold in the pre-state since it has to be an owner (by the T-Call rule of Figure 5.1 and predicate *invariantOk* of Figure 5.4) >

T = *typeOf*(new T(*e*)) ∧ {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

∧ (D1 ∉ *TypeId* ∨ S<*e* == null> ∨ S<*inv*(D1)[this←*e*]>)

⇒ < since there are no calls during the execution of *rcvr*=new T(*e*), the state of object *e* cannot be changed, so the invariant of object *e* must hold in the post-state S′ >

T = *typeOf*(new T(*e*)) ∧ {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[this←*e*]>)

⇒ < since there are no calls during the execution of *rcvr*=new T(*e*), the correctness proof of method T.T must be valid since all of the statements executed preserve validity (Lemmas 5.49 - 5.60) >

T = *typeOf*(new T(*e*)) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[this←*e*]>)

⇒ < by the Constructor Call Lemma 5.66 and since the run-time type D = T >

S′<*inv*(D)[this←*rcvr*] && *ens*(T, *m*)[this←*rcvr*]>

∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[this←*e*]>)

Case 5: *m*(*e*) is a superclass constructor call, e.g., super(*e*)

T = *typeOf*(super) ∧ {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

⇒ < from our assumption, the run-time type invariant of *e* (if it references an object) must hold in the pre-state (it would also have to be an owner variable) >

T = *typeOf*(super) ∧ {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

∧ (D1 ∉ *TypeId* ∨ S<*e* == null> ∨ S<*inv*(D1)[this←*e*]>)

⇒ < since there are no calls during the execution of super(*e*), the state of object *e* cannot be changed, so the invariant of object *e* must hold in the post-state S′ >

T = *typeOf*(super) ∧ {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}

∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[this←*e*]>)

⇒ < since there are no calls during the execution of super(*e*), the correctness proof of method T.T must be valid since all of the statements executed preserve validity (Lemmas 5.49 - 5.60) >

T = *typeOf*(`super`) ∧ |= {*req*(T, T)} *getBody*(T, T) {*inv*(T) && *ens*(T, T)}
∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[`this`←*e*]>)
⇒ < by the Superclass Constructor Lemma 5.67 and, from the perspective of
the constructor, the run-time type D = T; also superclass constructor calls can
only occur as the first statement in a subclass constructor and the subclass
constructor must establish the run-time type invariant of the object being initialized >
S′<*inv*(D) && *ens*(T, *m*)>
∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[`this`←*e*]>)
⇒ < `this` is the receiver in a superclass constructor call so we substitute `this` for `this`
which does not change the value of the expression >
S′<*inv*(D)[`this`←`this`] && *ens*(T, *m*)[`this`←`this`]>
∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[`this`←*e*]>)

Induction Step: Let k = N.
The induction hypothesis asserts that
If (∀ T1 ∈ *TypeId*, *n* ∈ *MethId* :
{*inv*(T1) && *req*(T1, *n*)} *getBody*(T1, *n*) {*inv*(T1) && *ens*(T1, *n*)})
∧ (∀ T1 ∈ *TypeId* :
{*req*(T1, T1)} *getBody*(T1, T1) {*inv*(T1) && *ens*(T1, T1)}),
∧ *k* < N ∧ <T.*m*, U1.*n1*, U2.*n2*, ... , Uk.*nk*> is a valid call chain,
then S′<*inv*(D)[`this`←*rcvr*] && *ens*(T, *m*)[`this`←*rcvr*]>
∧ (D1 ∉ *TypeId* ∨ S′<*e* == null> ∨ S′<*inv*(D1)[`this`←*e*]>).

If each method directly called by T.*m* preserves the validity of its specification, then T.*m* will also preserve the validity of its specification because then every statement in T.*m* will preserve validity. That is, it suffices to show that each direct call U1.*n1* made from T.*m* in a valid call chain preserves validity based on the induction hypothesis.

Without loss of generality, we only need to consider the first call made by T.*m* since the post-state after each call will establish the necessary preconditions for subsequent calls, i.e., that the run-time type invariant of argument objects hold in the pre- and post-states before and after the execution of T.*m*. Furthermore, a method can only invalidate the invariant of its receiver since our technique does not allow assignment to fields of objects other than the receiver. Therefore, method calls must re-establish the invariant of the receiver in the post-state. Again, there are five cases.

Case 1: U1.*n1* is an object-call

Since this is the first call, only the invariant of the current receiver among visible owner variables can be false since our technique only allows direct assignment to local variables and fields of the receiver. Also, if the object-call of U1.*n1* is a this-argument call, then, by the A-ObjCall rule of Figure

4.21, the invariant of the current receiver has to be established before the call. Therefore, by the induction hypothesis, the call of U1.*n1* satisfies its specification, i.e., preserves the validity of its specification; thus, on exit, it re-establishes the run-time type invariant of its argument objects (by the Object-Call Lemma 5.65).

Case 2: U1.*n1* is a self-call

Since we are considering the first call made by T.*m*, only the invariant of the current receiver among visible owner variables can be false (as explained in Case 1). However, before the self-call of U1.*n1*, the invariant of the current receiver must be established in accordance with the A-SelfCall rule of Figure 4.21. Therefore, by the induction hypothesis, the self-call of U1.*n1* satisfies its specification and, on exit, it re-establishes the run-time type invariant of its argument objects (by the Self-Call Lemma 5.63).

Case 3: U1.*n1* is a super-call

Since we are considering the first call made by T.*m*, only the invariant of the current receiver among visible owner variables can be false (as explained in Case 1). However, before the super-call of U1.*n1*, the invariant of the current receiver must be established in accordance with the A-SupCall rule of Figure 4.21. Therefore, by the induction hypothesis, the super-call of U1.*n1* satisfies its specification and, on exit, it re-establishes the run-time type invariant of its argument objects (by the Super-Call Lemma 5.65).

Case 4: U1.*n1* is a new object constructor call

Since we are considering the first call made by T.*m*, only the invariant of the current receiver among visible owner variables can be false (as explained in Case 1). However, if the new object constructor call of U1.*n1* is a this-argument call, then, by the A-NewAssign rule of Figure 4.22, the invariant of the current receiver has to be established before the call. Therefore, by the induction hypothesis, the call of U1.*n1* satisfies its specification and, on exit, it establishes the run-time type invariant of its argument objects (by the Constructor Call Lemma 5.66).

Case 5: U1.*n1* is a superclass constructor call

Since we are considering the first call made by T.*m*, only the invariant of the current receiver among visible owner variables can be false (as explained in Case 1). Furthermore, the superclass constructor call of U1.*n1* cannot be a this-argument call as restricted by the T-SupConstr rule of Figure 5.1. That is, the formal parameter cannot be an alias of its receiver because a constructor will not have initialized its fields or established its invariant; thus its formal parameter will have to be some owner variable other than the receiver, i.e., one with a valid invariant. Also, even though the invariant of the current receiver is not guaranteed to be established before a superclass constructor call, it does have to be established before that constructor can make a self-call or super-call; however, this does not establish the subclass invariant, so downcalls cannot be allowed. Therefore, a superclass constructor call is not allowed by our technique if it makes downcalls as specified in the T-SupConstr rule of Figure 5.1 and the *noDownCalls* predicate of Figure 5.4. Therefore, by the induction hypothesis, the

super-call of U1.*n1* satisfies its specification and, on exit, it establishes the run-time type invariant of its argument objects (by the Superclass Constructor Lemma 5.67). These calls can only occur as the first statement in a subclass constructor, so the calling subclass constructor establishes the subclass invariant after this call to ensure that the run-time type invariant of the receiver is established.

Based on these five cases, T.*m* satisfies its specification because all of its calls satisfy their specification; thus, on exit, based on the Self-Call Lemma 5.63, Super-Call Lemma 5.64, Object-Call Lemma 5.65, Constructor Call Lemma 5.66, and Superclass Constructor Lemma 5.66 (T.*m* must be one of these kinds of calls), the run-time type invariant of T.*m*'s argument objects are re-established when necessary.

■

Theorem 5.69 (Owner Invariant): Let O be the set of owner variables from the T-rules of Figures 5.1-5.3 at some point in method U.*n*. Let S ∈ *State* be an intermediate state at that same point during the execution of U.*n*. Let *vr* be a variable accessible at that same point in U.*n*. Let D be the run-time type of *vr*.

If D ∈ *TypeId* ∧ (*vr* ∈ O ∨ *vr* ≡ `p` ∨ *isPivot*(*vr*, U)) ∧ S<*vr* != null>,

then S<*inv*(D)[`this`←*vr*]>.

Proof:

Object *vr* can only be modified through an object-call. Thus the proof will be by induction on the number of object-calls made on *vr*.

Basis: *k* = 0, i.e., there are no object-calls on *vr*.

When object *vr* was created and initialized by a constructor, the run-time type invariant for *vr* was established; thus the invariant will continue to hold in S, i.e., S<*inv*(D)[`this`←*vr*]> holds in methods and constructors where *vr* is an owner variable other than the receiver `this`.

Induction Step: Let k = N.

The induction hypothesis asserts that if the number of object-calls on *vr* is less than N, then its invariant will hold in S, i.e., S<*inv*(D)[`this`←*vr*]>. So consider the Nth object-call on *vr*. The invariant holds prior to the Nth call, so it must also hold after that object-call by the Method Call Theorem 5.68.

■

The Owner Invariant Theorem 5.69 proves that, in our technique, the run-time type invariant of an object referenced by an owner variable *vr*, other than the receiver, holds everywhere that *vr* is visible.

5.2.6.6 Method invocation rules preserve validity

Using the Method Call Theorem 5.68, we now prove that the method invocation rules preserve validity.

Lemma 5.70 (A-SelfCall): The A-SelfCall rule of Figure 4.21 preserves validity

Proof:

Let S ∈ *State*. If S<*inv*(U) && *req*(T, *m*)[p←*e*]> = false or {`this.`*m*(*e*), S} does not terminate, then the A-SelfCall rule is trivially valid. So assume S<*inv*(U) && *req*(T, *m*)[p←*e*]> = true and that {`this.`*m*(*e*), S} terminates. Let D be the run-time type of the receiver.

S<*inv*(U) && *req*(T, *m*)[p←*e*]> ∧ U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*)
∧ {`this.`*m*(*e*), S} ⇒$_s$ S′

⇒ < establishing *inv*(U) in a method declared in U establishes the run-time type
invariant of the receiver (by the Valid Invariant Theorem 5.30) >

S<*inv*(D) && *req*(T, *m*)[p←*e*]> ∧ U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(U, *m*)
∧ {`this.`*m*(*e*), S} ⇒$_s$ S′

⇒ < by the Owner Invariant Theorem 5.69, the invariant of the receiver is the only owner
variable in the current context that could reference an object with an invalid invariant;
thus by the Method Call Theorem 5.68 >

S′<*inv*(D) && *ens*(T, *m*)>

⇒ < by the Invariant Clause Lemma 5.1 (in case D < U) >

S′<*inv*(U) && *ens*(T, *m*)>

■

Lemma 5.71 (A-SupCall): The A-SupCall rule of Figure 4.21 preserves validity

Proof:

Let S ∈ *State*. If S<*inv*(U) && *req*(T, *m*)[p←*e*]> = false or {`super.`*m*(*e*), S} does not terminate, then the A-SupCall rule is trivially valid. So assume S<*inv*(U) && *req*(T, *m*)[p←*e*]> = true and that {`super.`*m*(*e*), S} terminates. Let D be the run-time type of the receiver.

S<*inv*(U) && *req*(T, *m*)[p←*e*]> ∧ U = *typeOf*(`this`)
∧ T = *whereMethodDecl*(*superOf*(U), *m*) ∧ {`super.`*m*(*e*), S} ⇒$_s$ S′

⇒ < establishing *inv*(U) in a method declared in U establishes the run-time type
invariant of the receiver (by the Valid Invariant Theorem 5.30) >

S<*inv*(D) && *req*(T, *m*)[p←*e*]> ∧ U = *typeOf*(`this`)
∧ T = *whereMethodDecl*(*superOf*(U), *m*) ∧ {`super.`*m*(*e*), S} ⇒$_s$ S′

⇒ < by the Owner Invariant Theorem 5.69, the invariant of the receiver is the only owner
variable in the current context that could reference an object with an invalid invariant;
thus by the Method Call Theorem 5.68 >

S′<*inv*(D) && *ens*(T, *m*)>

⇒ < by the Invariant Clause Lemma 5.1 (in case D < U) >

S′<*inv*(U) && *ens*(T, *m*)>

■

Lemma 5.72 (A-ObjCall): The A-ObjCall rule of Figure 4.21 preserves validity

Proof:

Let $\mathsf{S} \in \mathit{State}$. If $\mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}, \texttt{p}\leftarrow e]\rangle$ = false or $\{\mathit{vr}\,.\,m(e), \mathsf{S}\}$ does not terminate, then the A-ObjCall rule is trivially valid. So assume $\mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}, \texttt{p}\leftarrow e]\rangle$ = true and that $\{\mathit{vr}\,.\,m(e), \mathsf{S}\}$ terminates.

Case 1: ($e \equiv$ `this`)

Let D1 be the run-time type of object *vr* and let D be the run-time type of the current receiver `this`.

$\mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}, \texttt{p}\leftarrow \texttt{this}]\rangle \wedge \mathrm{U} = \mathit{typeOf}(\texttt{this})$
$\wedge\ \mathrm{T} = \mathit{whereMethodDecl}(\mathit{typeOf}(\mathit{vr}), m) \wedge \{\mathit{vr}\,.\,m(\texttt{this}), \mathsf{S}\} \Rightarrow_s \mathsf{S}'$
$\Rightarrow$ < since *vr* is not `this`, its run-time type invariant must hold (by the Owner Invariant Theorem 5.69 and since it must be an owner variable) >
$\mathsf{S}\langle \mathit{inv}(\mathrm{D1})[\texttt{this}\leftarrow \mathit{vr}]\rangle \wedge \mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}, \texttt{p}\leftarrow \texttt{this}]\rangle \wedge \mathrm{U} = \mathit{typeOf}(\texttt{this})$
$\wedge\ \mathrm{T} = \mathit{whereMethodDecl}(\mathit{typeOf}(\mathit{vr}), m) \wedge \{\mathit{vr}\,.\,m(\texttt{this}), \mathsf{S}\} \Rightarrow_s \mathsf{S}'$
$\Rightarrow$ < when $e \equiv$ `this`, *inv*(U) must be established (by the A-ObjCall rule of Figure 4.21); since *inv*(U) is established in a method declared in U, this also establishes the run-time type invariant of the current receiver and *e* (by the Valid Invariant Theorem 5.30) >
$\mathsf{S}\langle \mathit{inv}(\mathrm{D1})[\texttt{this}\leftarrow \mathit{vr}]\rangle \wedge \mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}, \texttt{p}\leftarrow \texttt{this}]\rangle \wedge \mathrm{U} = \mathit{typeOf}(\texttt{this})$
$\wedge\ \mathrm{T} = \mathit{whereMethodDecl}(\mathit{typeOf}(\mathit{vr}), m) \wedge \{\mathit{vr}\,.\,m(\texttt{this}), \mathsf{S}\} \Rightarrow_s \mathsf{S}'$
$\wedge\ \mathsf{S}\langle \mathit{inv}(\mathrm{D})\rangle$
$\Rightarrow$ < thus there can be no owner variable visible in *m* with an invalid invariant (by the Owner Invariant Theorem 5.69); thus by the Method Call Theorem 5.68 >
$\mathsf{S}'\langle \mathit{inv}(\mathrm{D1})[\texttt{this}\leftarrow \mathit{vr}]$ && $\mathit{ens}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}]\rangle$
$\Rightarrow$ < semantics of the && operator and logic >
$\mathsf{S}'\langle \mathit{ens}(\mathrm{T}, m)[\texttt{this}\leftarrow \mathit{vr}]\rangle$

Case 2: !($e \equiv$ `this`)

Let $\mathsf{S}' \in$ State be the post-state. Since *vr* and *e* are not the current receiver `this`, their run-time type invariants must hold (by the Owner Invariant Theorem 5.69 and since both must be owner variables). Thus there can be no owner variable visible in the called method *m* with an invalid invariant. Hence, by the Method Call Theorem 5.68, $\mathsf{S}'\langle \mathit{ens}(\mathrm{T}, m)\rangle$.

■

Lemma 5.73 (A-NewAssign): The A-NewAssign rule of Figure 4.22 preserves validity

Proof:

Let $\mathsf{S} \in \mathit{State}$. If $\mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{p}\leftarrow e]\rangle$ = false or $\{\mathit{vr}=\texttt{new}\ \mathrm{T}(e), \mathsf{S}\}$ does not terminate, then the A-NewAssign rule is trivially valid. So assume $\mathsf{S}\langle \mathit{req}(\mathrm{T}, m)[\texttt{p}\leftarrow e]\rangle$ = true and that $\{\mathit{vr}=\texttt{new}\ \mathrm{T}(e), \mathsf{S}\}$ terminates.

Case 1: ($e \equiv$ `this`)

Let D be the run-time type of the current receiver `this`.

S<*req*(T, *m*)[p←`this`]> ∧ U = *typeOf*(`this`) ∧ {*vr*=`new` T(`this`), S} ⇒$_s$ S′

⇒ < when $e \equiv$ `this`, *inv*(U) must be established (by the A-NewAssign rule of Figure 4.22); since *inv*(U) is established in a method declared in U, this also establishes the run-time type invariant of the current receiver (by the Valid Invariant Theorem 5.30) >

S<*req*(T, *m*)[p←`this`]> ∧ U = *typeOf*(`this`) ∧ {*vr*=`new` T(`this`), S} ⇒$_s$ S′
∧ S<*inv*(D)>

⇒ < thus there can be no owner variable visible in T . T with an invalid invariant (since no pivot fields have been initialized yet); thus by the Method Call Theorem 5.68 >

S′<*inv*(T)[`this`←*vr*] && *ens*(T, *m*)[`this`←*vr*]>

Case 2: !($e \equiv$ `this`)

Let {*vr*=`new` T(`this`), S} ⇒$_s$ S′. Since *e* is not `this`, its run-time type invariant must hold (by the Owner Invariant Theorem 5.69 and since it must be an owner variable). Thus there can be no owner variable visible in *m* with an invalidated invariant. Hence, by the Method Call Theorem 5.68, S′<*inv*(T)[`this`←*vr*] && *ens*(T, *m*)[`this`←*vr*]>.

■

Lemma 5.74 (A-SupConstr): The A-SupConstr rule of Figure 4.21 preserves validity

Proof:

Let S ∈ *State*. If S<*req*(T, *m*)[p←*e*]> = false or {`super`(*e*), S} does not terminate, then the A-SupConstr rule is trivially valid. So assume S<*req*(T, *m*)[p←*e*]> = true and that {`super`(*e*), S} terminates. Let D1 be the run-time type of *e*.

S<*req*(T, *m*)[p←*e*]> ∧ U = *typeOf*(`this`) ∧ T = *superOf*(U) ∧ {`super`(*e*), S} ⇒$_s$ S′

⇒ < when !($e \equiv$ `this`), *e*'s run-time type invariant must hold if *e* is an object reference (by the Owner Invariant Theorem 5.69 and since it must be an owner variable) >

S<*req*(T, *m*)[p←*e*]> ∧ U = *typeOf*(`this`) ∧ T = *superOf*(U) ∧ {`super`(*e*), S} ⇒$_s$ S′
∧ (D1 ∉ *TypeId* ∨ S<*e* == null> ∨ S<*inv*(D1)[`this`←*e*]>)

⇒ < thus there can be no owner variables visible in T . T with an invalid invariant (since no pivot fields have been initialized yet); thus by the Method Call Theorem 5.68 >

S′<*inv*(T) && *ens*(T, *m*)>

■

Our next lemma will be used to simplify the proofs of the two lemmas that follow; it proves that, when a method call is the right side of an assignment statement, we can substitute the target variable of the assignment for the JML pseudo variable `\result` in the postcondition (`\result` denotes the return value in the postconditions of JML method specifications).

Lemma 5.75(Result Substitution): Let $S \in State$.

If $S\langle \mathbf{P} \rangle \wedge [vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v \wedge vr$ does not occur in **P**,
then $S[vLoc := v]\langle \mathbf{P}[\texttt{\result} \leftarrow vr] \rangle$.
Proof:

$S\langle \mathbf{P} \rangle \wedge [vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v$
$\Rightarrow <$ by the Substitution Theorem 5.9 $>$
$S\langle \mathbf{P} \rangle \wedge [vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v$
$\wedge\ S\langle \mathbf{P}[vr \leftarrow \texttt{\result}] \rangle = S[vLoc := v]\langle \mathbf{P} \rangle$
$\Rightarrow <$ $\mathbf{P} \equiv \mathbf{P}[vr \leftarrow \texttt{\result}]$ because vr does not occur in **P** $>$
$S\langle \mathbf{P} \rangle \wedge [vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v$
$\wedge\ S\langle \mathbf{P} \rangle = S[vLoc := v]\langle \mathbf{P} \rangle$
$\Rightarrow <$ $S\langle \mathbf{P} \rangle$ and equality $>$
$[vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v \wedge S[vLoc := v]\langle \mathbf{P} \rangle$
$\Rightarrow <$ from the E-VarRef rule of Figure 4.16 $>$
$[vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v \wedge S[vLoc := v]\langle \mathbf{P} \rangle$
$\wedge\ [vr, S[vLoc := v]] \Rightarrow_{e} getValue(S[vLoc := v], vLoc)$
$\Rightarrow <$ by the State Update Lemma 5.6 $>$
$[vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S] \Rightarrow_{e} v \wedge S[vLoc := v]\langle \mathbf{P} \rangle$
$\wedge\ [vr, S[vLoc := v]] \Rightarrow_{e} v$
$\Rightarrow <$ we can substitute equals for equals (when expressions do not have side-effects)
without changing the expression's value, i.e., $[\texttt{\result} \leftarrow vr]$ $>$
$S[vLoc := v]\langle \mathbf{P}[\texttt{\result} \leftarrow vr] \rangle$

■

Lemma 5.76(A-CallAssign): The A-CallAssign rule of Figure 4.22 preserves validity

Proof:

Let $S \in State$. If $S\langle \mathbf{P} \rangle$ = false or $\{vr=e0.m(e), S\}$ does not terminate, then the A-CallAssign rule is trivially valid. So assume $S\langle \mathbf{P} \rangle$ = true and that $\{vr=e0.m(e), S\}$ terminates.

We start calculating from our assumptions and the semantics of $vr=e0.m(e)$ given in the S-CallAssign rule of Figure 4.19.

$S\langle \mathbf{P} \rangle \wedge \{e0.m(e), S\} \Rightarrow_{s} S' \wedge [vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S'] \Rightarrow_{e} v$
$\wedge\ \{vr=e0.m(e), S\} \Rightarrow_{s} S'[vLoc := v]$
$\Rightarrow <$ by the validity of $\{\mathbf{P}\}\ e0.m(e)\ \{\mathbf{Q}\}$ in the antecedent of the A-CallAssign rule of Figure 4.22 $>$
$[vr, S] \Rightarrow_{lv} vLoc \wedge [\texttt{\result}, S'] \Rightarrow_{e} v \wedge \{vr=e0.m(e), S\} \Rightarrow_{s} S'[vLoc := v]$
$\wedge\ S'\langle \mathbf{Q} \rangle$
$\Rightarrow <$ by the Result Substitution Lemma 5.75 since vr is not in scope in **Q** $>$

S′[vLoc := v]<**Q**[\result←*vr*]>

■

Lemma 5.77(A-SupCallAssign): The A-SupCallAssign rule of Figure 4.22 preserves validity

Proof:

Let S ∈ *State*. If S<**P**> = false or {*vr*=`super.`*m*(*e*), S} does not terminate, then the A-SupCallAssign rule is trivially valid. So assume S<**P**> = true and that {*vr*=`super.`*m*(*e*), S} terminates.

We start calculating from our assumptions and the semantics of *vr*=`super.`*m*(*e*) given in the S-SupCallAssign rule of Figure 4.19.

S<**P**> ∧ {`super.`*m*(*e*), S} ⇒$_s$ S′ ∧ [*vr*, S] ⇒$_{lv}$ vLoc ∧ [\result, S′] ⇒$_e$ v
∧ {*vr*=`super.`*m*(*e*), S} ⇒$_s$ S′[vLoc := v]

⇒ < by the validity of {**P**} `super.`*m*(*e*) {**Q**} in the antecedent of the A-CallAssign rule of Figure 4.22 >

[*vr*, S] ⇒$_{lv}$ vLoc ∧ [\result, S′] ⇒$_e$ v ∧ {*vr*=`super.`*m*(*e*), S} ⇒$_s$ S′[vLoc := v]
∧ S′<**Q**>

⇒ < by the Result Substitution Lemma 5.75 since *vr* is not in scope in **Q** >

S′[vLoc := v]<**Q**[\result←*vr*]>

■

5.2.6.7 The assignable rules preserve validity

Our last two lemmas show that the A-Assignable and A-SupAssignable rules preserve validity. These rules can be used by verifiers when reasoning about side-effects, particularly when variables are not mentioned in the postcondition of the called method, i.e., these inference rules allow the verifier to prove that variables did not change during a method call. The A-SupAssignable rule can be used in reasoning about assignments to subclass fields that overriding subclass methods can assign to. That is, the A-SupAssignable rule can be used to prove that a super-call does not have unverifiable additional side-effects; of course, this rule assumes that the super-call is allowed by our rules.

Lemma 5.78 (A-SupAssignable): The A-SupAssignable rule of Figure 4.23 preserves validity

Proof:

Let S ∈ *State*. If for all *Z* ∈ *LogicId*, S<**P** && *Z*==*w*> = false or {`super.`*m*(*e*), S} does not terminate, then the A-SupAssignable rule is trivially valid. So assume S<**P** && *Z*==*w*> = true and {`super.`*m*(*e*), S} terminates.

S<**P** && *Z*==*w*> ∧ U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(*superOf*(U), *m*)
∧ U ≤ T1 < T ∧ *f* ∈ *setOfFieldsIn*(T1) ∧ (*w* ≡ `this.`*f* ∨ *w* ≡ `this.`*f*`.`*g*)
∧ {`super.`*m*(*e*), S} ⇒$_s$ S′

⇒ < logic and meaning of S<**P** && *Z*==*w*> >

S<**P**> ∧ S<*Z*==*w*> ∧ U = *typeOf*(`this`) ∧ T = *whereMethodDecl*(*superOf*(U), *m*)

$\wedge$ U $\leq$ T1 < T $\wedge$ $f \in$ *setOfFieldsIn*(T1) $\wedge$ ($w \equiv$ `this.`$f \vee w \equiv$ `this.`f`.`g)

$\wedge$ {`super.`$m(e)$, S} $\Rightarrow_s$ S′

$\Rightarrow$ < *Z* is a logical variable and thus is independent of the program state >

S<**P**> $\wedge$ *Z*==S<*w*> $\wedge$ U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*)

$\wedge$ U $\leq$ T1 < T $\wedge$ $f \in$ *setOfFieldsIn*(T1) $\wedge$ ($w \equiv$ `this.`$f \vee w \equiv$ `this.`f`.`g)

$\wedge$ {`super.`$m(e)$, S} $\Rightarrow_s$ S′

$\Rightarrow$ < |= {**P**} `super.`$m(e)$ {**Q**}, since it is an antecedent of the A-SupAssignable rule of Figure 4.23 >

S′<**Q**> $\wedge$ *Z*==S<*w*> $\wedge$ U = *typeOf*(`this`) $\wedge$ T = *whereMethodDecl*(*superOf*(U), *m*)

$\wedge$ U $\leq$ T1 < T $\wedge$ $f \in$ *setOfFieldsIn*(T1) $\wedge$ ($w \equiv$ `this.`$f \vee w \equiv$ `this.`f`.`g)

$\wedge$ {`super.`$m(e)$, S} $\Rightarrow_s$ S′

$\Rightarrow$ < by the Additional Side-Effects Theorem 5.48, since `super.`$m(e)$ is allowed by our rules >

S′<**Q**> $\wedge$ *Z*==S<*w*> $\wedge$ S<*w*> = S′<*w*>

$\Rightarrow$ < equality >

S′<**Q**> $\wedge$ *Z*==S′<*w*>

$\Rightarrow$ < *Z* is a logical variable and thus is independent of the program state >

S′<**Q**> $\wedge$ S′<*Z*==*w*>

$\Rightarrow$ < by the meaning of the && operator and logic >

S′<**Q** && *Z*==*w*>

■

Lemma 5.79 (A-Assignable): The A-Assignable rule of Figure 4.23 preserves validity

Proof:

Let S $\in$ *State*. If for all *Z* $\in$ *LogicId*, S<**P** && *Z*==*w*> = false or {*e0*`.`*m*(*e*), S} does not terminate, then the A-Assignable rule is trivially valid. So assume S<**P** && *Z*==*w*> = true and {*e0*`.`*m*(*e*), S} terminates.

S<**P** && *Z*==*w*> $\wedge$ U = *typeOf*(*e0*) $\wedge$ T = *whereMethodDecl*(U, *m*) $\wedge$ U $\leq$ T

$\wedge$ {*e0*`.`*m*(*e*), S} $\Rightarrow_s$ S′

$\wedge$ $w \notin_a$ *selfAssigns*(*e0*, T, *m*) $\wedge$ $w \notin_a$ *parmAssigns*(*e*, T, *m*)

$\Rightarrow$ < definition of *selfAssigns* and *parmAssigns* of Figure 5.4 >

S<**P** && *Z*==*w*> $\wedge$ U = *typeOf*(*e0*) $\wedge$ T = *whereMethodDecl*(U, *m*) $\wedge$ U $\leq$ T

$\wedge$ {*e0*`.`*m*(*e*), S} $\Rightarrow_s$ S′

$\wedge$ $w \notin_a$ { `this.`f | `this.`$f \in$ *assigns*(T, *m*) }[`this`$\leftarrow$*e0*]

$\wedge$ $w \notin_a$ { `this.`f`.`g | `this.`f`.`$g \in$ *assigns*(T, *m*) }[`this`$\leftarrow$*e0*]

$\wedge$ $w \notin_a$ { `p.`g | `p.`$g \in$ *assigns*(T, *m*) }[`p`$\leftarrow$*e*]

$\Rightarrow$ < definition of set membership and substitution and since the assignable clause can only reference fields of the receiver or formal parameter (see JML syntax and the

predicate *validMethodSpecs* of Figure 5.7) >

S<**P** && *Z*==*w*> ∧ U = *typeOf*(*e0*) ∧ T = *whereMethodDecl*(U, *m*) ∧ U ≤ T

∧ {*e0*.*m*(*e*), S} ⇒$_s$ S′ ∧ *w* ∉$_a$ *assigns*(T, *m*)[this←*e0*, p←*e*]

⇒ < logic and meaning of S<**P** && *Z*==*w*> >

S<**P**> ∧ S<*Z*==*w*> ∧ U = *typeOf*(*e0*) ∧ T = *whereMethodDecl*(U, *m*) ∧ U ≤ T

∧ {*e0*.*m*(*e*), S} ⇒$_s$ S′ ∧ *w* ∉$_a$ *assigns*(T, *m*)[this←*e0*, p←*e*]

⇒ < *Z* is a logical variable and thus is independent of the program state >

S<**P**> ∧ *Z*==S<*w*> ∧ U = *typeOf*(*e0*) ∧ T = *whereMethodDecl*(U, *m*) ∧ U ≤ T

∧ {*e0*.*m*(*e*), S} ⇒$_s$ S′ ∧ *w* ∉$_a$ *assigns*(T, *m*)[this←*e0*, p←*e*]

⇒ < |= {**P**} *e0*.*m*(*e*) {**Q**}, since it is an antecedent of the A-Assignable rule

of Figure 4.23 >

S′<**Q**> ∧ *Z*==S<*w*> ∧ U = *typeOf*(*e0*) ∧ T = *whereMethodDecl*(U, *m*) ∧ U ≤ T

∧ {*e0*.*m*(*e*), S} ⇒$_s$ S′ ∧ *w* ∉$_a$ *assigns*(T, *m*)[this←*e0*, p←*e*]

⇒ < by the Assignable Clause Theorem 5.43, since *e0*.*m*(*e*) is allowed by our rules >

S′<**Q**> ∧ *Z*==S<*w*> ∧ S<*w*> = S′<*w*>

⇒ < equality >

S′<**Q**> ∧ *Z*==S′<*w*>

⇒ < *Z* is a logical variable and thus is independent of the program state >

S′<**Q**> ∧ S′<*Z*==*w*>

⇒ < by the meaning of the && operator and logic >

S′<**Q** && *Z*==*w*>

■

5.3 Discussion, Conclusions, and Future Work

In this chapter, we formalized our rules from Chapters 2 and 3 so they could be used in our soundness proof. However, the T-rules and their associated predicates and functions are sometimes more conservative than the informal rules from Chapters 2 and 3; in particular, the formalization of two of our rules is more restrictive than the informal versions. In this section, we examine how the original less conservative versions of these two rules could be formalized and give an informal justification of their soundness. Recall that we chose the more conservative versions to simplify the soundness proof given in Section 5.2.

In subsection 5.3.1, we reconsider the formalization of the Invariant Invalidation Rule from Chapter 2. Similarly, in subsection 5.3.2, we revisit the Actual Parameter Aliasing Rule from Chapter 3. In subsection 5.3.3, we examine what would be necessary to eliminate our assumption and requirement that unoverrideable methods not have side-effects. In subsection 5.3.4, we discuss what would be involved in extending our formalization to methods with more than one formal parameter. In subsection 5.3.5, we discuss ways of making our handling of object-calls in the callable clause less

$$validInvariant(\mathrm{S}, \mathrm{T}, m) = noSelfCalls(\mathrm{T}, m) \wedge$$
$$(\forall f \in allFieldsIn(\mathrm{T}), g \in VarId :$$
$$(\texttt{this}.f \notin_{\mathrm{a}} assigns(\mathrm{T}, m) \vee \texttt{this}.f \notin accessed(invOf(\texttt{TEnv}(\mathrm{S}))))$$
$$\wedge (\texttt{this}.f.g \notin_{\mathrm{a}} assigns(\mathrm{T}, m) \vee \texttt{this}.f.g \notin accessed(invOf(\texttt{TEnv}(\mathrm{S})))))$$

$$noSelfCalls(\mathrm{T}, m) = \{ \texttt{this}.m \mid \texttt{this}.m \in calls(\mathrm{T}, m) \} = \{ \}$$
$$\wedge noParmSelfCalls(\mathrm{T}, m, \mathrm{U}, n)$$
$$\wedge (\forall \mathrm{U}::n \in calls(\mathrm{T}, m) : noSelfCalls(\mathrm{U}, n))$$

$$noParmSelfCalls(\mathrm{T}, \mathrm{U}, n) =$$
$$getParmType(\mathrm{U}, n) \notin TypeId \vee !(\mathrm{T} \leq getParmType(\mathrm{U}, n))$$
$$\vee (\forall \mathrm{U1}::n1 \in calls(\mathrm{U}, n) : !(\mathrm{U1} \leq getParmType(\mathrm{U}, n)))$$

$$aliasingOK(rcvr, arg, \mathrm{U}, \mathrm{T}, m) = !(rcvr \equiv arg)$$
$$\wedge (!(rcvr \equiv \texttt{this}) \vee !isPivot(arg, \mathrm{U})$$
$$\vee (\forall g \in VarId : arg.g \notin_{\mathrm{a}} assigns(\mathrm{T}, m)))$$
$$\wedge (!(arg \equiv \texttt{this}) \vee !isPivot(rcvr, \mathrm{U})$$
$$\vee (\forall g \in VarId : rcvr.g \notin_{\mathrm{a}} assigns(\mathrm{T}, m)))$$

$$parmAliasingOK(arg1, arg2) = !(arg1 \equiv arg2)$$

Figure 5.8: Less restrictive predicates that could be used in the T-Rules and by other functions in Figure 5.4.

conservative. Finally, in subsection 5.3.6, we discuss the practicality of our technique and how it would be used.

5.3.1 The Invariant Invalidation Rule

The version of the Invariant Invalidation Rule that we formalized in Section 5.1 via the predicate *validInvariant* disallows a super-call or requires an override whenever the superclass method is allowed to assign to fields that are constrained by a subclass invariant. Therefore, superclass methods cannot be called if they invalidate any of the subclass parts of the type invariant; thus the run-time type invariant is established whenever the invariant of the static type of the receiver is established (see the Valid Invariant Theorem 5.30).

However, the informal rule given in Chapter 2 (subsection 2.4.1) is less conservative in that it only disallows the super-call when the superclass method makes a self-call and has permission to assign to a superclass field that is constrained by a subclass invariant. Thus the informal rule is different in two

ways, i.e., the field that is allowed to change must be visible in the superclass and the superclass method must have permission to make a self-call. The reason this informal rule is sound is because super-calls cannot assign to subclass fields unless they make a self-call down to a subclass method; thus the super-call can only invalidate the subclass invariant if it assigns to fields visible in the superclass. Furthermore, if the super-call does not make a self-call (downcall), then the run-time type invariant does not have to be established during the super-call; the subclass invariant can be established by the calling subclass method. Note that we have to prevent self-calls, not just downcalls, because some other subclass lower in the hierarchy than the current one may override the method being self-called but the run-time type invariant would not hold. This less restrictive version of the Invariant Invalidation Rule as given in Chapter 2 is formalized by predicate *validInvariant* of Figure 5.8.

5.3.2 The Actual Parameter Aliasing Rule

The Actual Parameter Aliasing Rule was formalized by the T-rules in Section 5.1 via the *aliasingOk* predicate of Figure 5.4. However, this predicate is more conservative than the original informal rule given in Chapter 3. Recall that the informal rule requires the use of the accessible clause, that is, when a called method has side-effects, its argument objects are allowed to have shared state as long as the assignable fields are not assigned through one access path and read through another.

However, the use of the accessible clause is only sound if field accesses are only allowed through a unique access path, e.g., through an owner variable[6]. Also, since concrete fields are not visible in the public specification, accesses would have to be specified through data groups as is done when specifying side-effects. Furthermore, accessing fields of objects other than the receiver would have to be done through object-calls and the receiver would have to be an owner variable. Requiring that accesses to fields of objects other than the current receiver be done through object-calls is already a restriction in our technique for internal objects, i.e., based on the Predicate Clause Access Rule of subsection 3.3.4, an object-call on a field of the receiver means that that field must be a pivot. Note, however, that fields of newly created objects would not have to be specified or checked. What is unclear is whether the benefits of using our less conservative rule outweigh the additional overhead of specifying and checking the accessible clause. On the other hand, the accessible clause may be useful for other purposes.

Another, we believe, better possibility might be to only disallow two argument objects in a method call when they have overlapping fields (shared state) and the overlapping fields are assignable; this seems better since it does not require the specification or checking of field accesses. This modified rule is given below.

---

6. The soundness of the assignable clause depends on a similar restriction, i.e., side-effects can only be initiated through an owner variable.

Actual Parameter Aliasing Rule′. Two argument objects to a method call cannot have aliased fields if those fields are assignable in the called method.

The difference between the above rule and the version given in Chapter 3 is that the above rule does not allow the call even if the overlapping assignable fields are not accessed through a different access path. The above version (i.e., the Actual Parameter Aliasing Rule′) is formalized by predicate *aliasingOk* of Figure 5.8. This rule is sound because it prevents argument objects from having overlapping assignable fields (as in the No Overlapping Assignable Fields Lemma 5.25).

All three versions[7] of this rule have pros and cons, so a case study would be needed to determine which version is the most practical (by practical we mean the trade off between how many unnecessary errors are issued and the difficulty and efficiency of the implementation of each rule); we leave this case study as future work.

5.3.3 Unoverrideable Methods

We assumed in subsection 1.6.6 that unoverrideable methods with side-effects would not be allowed because such methods can cause subclasses to be unimplementable (see subsections 2.9.2 and 2.9.3). For example, if an unoverrideable method invalidates a subclass invariant, then the run-time type invariant probably would not be reestablished on exit (since it knows nothing about the subclass).

However, another approach would be to disallow a subclass invariant when that invariant cannot be established by all of its public and protected methods due to unoverrideable superclass methods. Thus the next rule could be added to our set of rules, replacing our assumption that disallows final methods with side-effects.

Subclass Invariant Rule: Let *S* be a subclass of *C*. Let *V* be a concrete instance variable visible in *C*. If superclass method *C*::*M* is unoverrideable and can assign to *V*, then *S* cannot specify an invariant that accesses and constrains the value of *V*.

This rule seems more practical than our original assumption because it would not require an error message when the superclass defines a final method with side-effects. Instead, this rule would only require an error message when a subclass declares an invariant that would be invalidated by an unoverrideable superclass method; in a sense, this is where the error actually occurs and it gives the customizer a chance to correct the error, i.e., remove the subclass invariant or remove final from the declaration of the unoverrideable method (if superclass code is available).

The above rule is similar to the usual restrictions on invariants found in the literature [PH97, Mül02, LM04, MPHL05, LM05]; however, the above rule would only restrict subclass invariants when the superclass has unoverrideable methods. That is, our technique does not impose this restriction except when there are unoverrideable superclass methods that may invalidate the subclass

7. The version formalized by *aliasingOk*, the version given in Chapter 3, and the version given in this subsection.

invariant because, in our technique, subclass invariants can be handled through method overrides and by disallowing super-calls.

However, the above rule does not eliminate the problems caused by additional side-effects. That is, even if a subclass does not declare a subclass invariant, the subclass will still be unimplementable if superclass code is unavailable and the subclass inherits unoverrideable methods with additional side-effects, i.e., the required assignments to subclass fields will not be possible and/or assignments to subclass fields through downcalls will be unverifiable without superclass code. Therefore, we need another rule to avoid such problems.

Additional Side-Effects Rule: Let *S* be a subclass of *C*. Let *V* be a concrete instance variable declared in *S*. If superclass method *C*::*M* is unoverrideable, then *S* cannot allow *C*::*M* to assign to *V*.

The above Additional Side-Effects Rule says that subclasses cannot add subclass fields to a data group that is assignable by an unoverrideable superclass method. Therefore, with the two rules given in this subsection, we would be able to eliminate our assumption (see subsection 1.6.6) that final methods do not have side-effects.

5.3.4 More than One Formal Parameter

For simplicity, we restricted the syntax of Java-C to methods with a single formal parameter. However, our technique can be extended to multiple formal parameters by checking each pair of actual parameters in a method call, i.e., the restrictions in *aliasingOK* would be applied to the receiver and each actual parameter and each pair of actual parameters could not be aliases. Therefore, there would have to be two functions, one for comparing the current receiver with each formal parameter as is done in Section 5.1 (using *aliasingOK*) and another for comparing each pair of non-receiver parameters (using *formalAliasingOK*). These predicates are given in Figure 5.8.

5.3.5 Object Calls in the Callable Clause

In our technique, the primary use of the callable clause is to identify self-calls and possible downcalls. However, because of visibility restrictions and aliasing, object-calls are specified in the callable clause with the type of the receiver rather than with the actual receiver expression. Therefore, our use of the callable clause has to be conservative because of aliasing. For example, we sometimes have to consider an object-call to be a self-call when the type of the receiver indicates that the object-call could be a self-call; this occurs primarily when trying to determine whether an object-call is also a this-argument call (see predicates *noParmSelfCalls* of Figure 5.8 and *noParmAddSideEffects* of Figure 5.5).

Nonetheless, we believe that identifying possible downcalls through the type of the formal parameters in an object-call will not make our technique impractical since most methods do not make this-argument calls (`this` is usually passed as the receiver in a self-call or super-call). Furthermore, our technique has restrictions that enable most self-calls to be identified, e.g, calls with side-effects

have to be initiated through the caller's receiver in a self-call or super-call or through an owner variable in an object-call; thus variables that initiate side-effects can only be the original owner or a parameter (also, local variable owners reference new objects and are not considered by our rules).

A similar approach to our use of the receiver type (rather than the receiver expression) was used in the enforcement of the Law of Demeter. That is, the Object Form of the Law of Demeter is undecidable, and thus cannot be statically enforced. However, the Class Form was successfully used and applied in a large project [LH89]. A case study of our technique for specifying and handling object-calls is needed but we leave that for future work.

However, another possible formalization that would make our technique less conservative would be to only require that self-calls, super-calls, and this-argument calls be listed in the callable clause since these are the only calls of interest in our technique (except when the Callback Cycle Rules are enforced). Therefore, object-calls would only be considered in the application of the T-rules when they are this-argument calls, rather than when they could be this-argument calls.

All of the rules that we formalized in Figure 5.8 and in Section 5.1 can be enforced when the callable clause only specifies the methods directly called; also, we are able to use the code contract that reflects the actual implementation code also making the T-rules more practical. However, the Callback Cycle Rules require that all directly or indirectly called methods be included in the callable clause since their purpose is to identify cycles in the call graph of a method; furthermore, to automate this would require that a tool calculate the transitive closure of the callable relation between methods as specified in the callable clause to determine the directly and indirectly called methods. We leave the enforcement of the Callback Cycle Rules as future work; thus our formalization only requires direct calls in the callable clause.

5.3.6 Our Technique, Its Limitations and Use

The enforcement of the rules of our technique may disallow some implementations that customizers may want and that may be safe. However, the soundness of our verification logic requires that these rules be enforced even though there may be exceptions where violations of our rules do not cause unexpected behavior or problems.

For example, the T-rules, with predicate *invariantOK* as an antecedent, are more restrictive than the Owner Variable Rule given in Chapter 3. That is, these T-rules require that all actual parameters be owner variables when they are a reference type, whereas in Chapter 3, this was only required when the called method had permission to change the state of the corresponding argument object. The reason for the more restrictive T-rules is because the only objects that are guaranteed not to have an invalidated type invariant are those referenced by owner variables; thus the more restrictive rule that we formalized is necessary for the soundness of our verification logic. However, if the verifier can prove that the type invariant holds for an object referenced by a non-owner variable (variables containing a

read-only reference), then it is safe to pass that object as a parameter as long as the called method does not change the state of that argument object.

Therefore, another way to view the enforcement of our rules is that they warn customizers of potential problems. In the above example, passing a non-owner variable as an actual parameter means that the verifier has to be sure that the run-time type invariant of such read-only references holds. This can only be guaranteed modularly if the owner of the object referenced by the read-only variable is visible in the local context [MPHL05]. However, one could imagine doing a whole program proof, but such a proof would be required again every time any of the classes is extended; furthermore, this proof would also require knowing every possible object (and its run-time type) that could be referenced by that non-owner variable.

What is unclear, without a case study, is how often implementers need to violate our rules; this is also left as future work.

The proof given in this chapter demonstrates that our technique is sound and can be used to avoid the problems caused by downcalls and aliasing. Furthermore, in Figure 5.8, we provided some replacement predicates that are less conservative than those given earlier; they also match the informal rules given in Chapters 2 and 3 and make our technique more practical than it would have been with the predicates given earlier.

CHAPTER 6: CLASS LIBRARY GUIDELINES AND TOOL SUPPORT

In this chapter, we present a set of guidelines for class library implementers; these guidelines make it easier for customizers to extend those library classes even when the superclass code is made available. We also give a brief description of the kind of tool support that would be needed to support our technique. The guidelines are presented in Section 6.1 and the description of the tool is given in Section 6.2. We conclude with a brief discussion in Section 6.3.

6.1 Class Library Guidelines

6.1.1 Introduction

A major purpose of our study of the problems caused by downcalls and aliasing was to identify the kinds of problems that can arise when creating a subclass without the superclass code. In particular, we wanted to see whether we could reason about the behavior of superclass methods using only the superclass specification. This led to the set of rules given in Chapters 2 and 3. That is, when a potential problem was identified, then rules or restrictions were given to prevent such problems. The rules are conservative but we believe that they identify issues that must be considered by OO programmers in general when creating subclasses; this is based on our experimentation with much more complicated examples than those given in Chapters 1, 2, and 3. Therefore, even if the superclass code is available, the calling structure, side-effects, and aliasing must be considered when determining which methods to override and when super-calls are safe. Furthermore, the pivot fields must be identified and the corresponding pivot objects must be protected to prevent unexpected side-effects, i.e., to make sure the abstract value of model fields cannot be changed unexpectedly and to ensure that the specification and use of the assignable clause is sound.

However, following these rules places a significant burden on the customizer extending classes in a framework or class library. Therefore, it seemed reasonable to also investigate whether adding a few more restrictions could ease this burden on customizers, i.e., make the class library a little more user friendly for customizers. For example, our study provides some insights for framework and class library designers and implementers that could eliminate the need to apply some or all of the rules given in Chapter 2 for dealing with downcalls. Thus the purpose of the guidelines described in subsection 6.1.2 is to make it easier for customizers to create subclasses.

One way to prevent the reasoning problems related to additional side-effects and subclass invariants is by requiring that library classes adhere to the calling structure shown in Figure 6.1; this calling structure greatly simplifies the reasoning required by customizers. The idea, as depicted in Figure 6.1, is to partition the methods of a class into three levels: overrideable non-pure, overrideable pure, and unoverrideable. Recall that a *pure* method is a method that has no side-effects. The direction of each arrow indicates that methods on one level can self-call the methods on the other level. Figure 6.1 describes the following calling structure:

1. overrideable methods with side-effects cannot be self-called by any methods of the class,

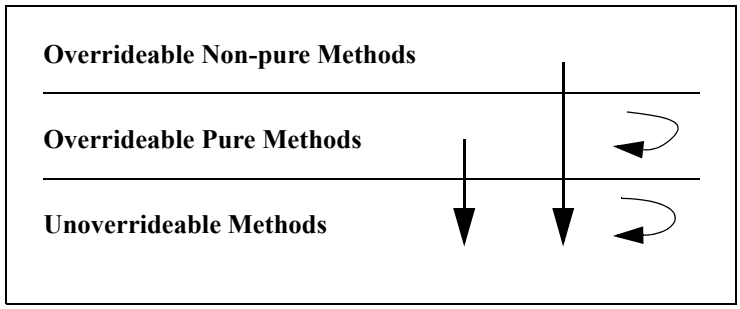

Figure 6.1: The three levels of methods used in the guidelines for library providers. Arrows indicate that calls can be made in the arrow's direction.

2. overrideable pure methods can only be self-called by other overrideable pure methods, and
3. unoverrideable methods can be freely called by all other methods of the class.

The overrideable methods of a class are the non-private methods that can be overridden by subclasses, i.e., the non-private, non-final methods in Java or the non-private, virtual methods in C++. However, our assumptions do not allow non-private methods to be unoverrideable if they have side-effects (see assumptions in subsection 1.6.6 and Guideline L6[1] below). That is, in our technique, unoverrideable methods with side-effects must all be private. Therefore, the only unoverrideable methods visible to clients and customizers must be pure, that is, final pure methods in Java or non-virtual pure methods in C++. In Smalltalk however, there are no unoverrideable methods; so, for Smalltalk, there would be no methods on level 3 of Figure 6.1.

The partitioning of methods into three levels and then restricting the calling structure of the superclass, as shown in Figure 6.1, prevents superclass methods from being invalidated by a new subclass if that subclass introduces additional side-effects or subclass invariants[2]. Therefore, a superclass method will be safe to super-call unless it calls down to a non-refining method (in JML non-refining methods are not allowed).

The reason super-calls will be safe is because this calling structure eliminates the possibility of downcalls while the type invariant is invalid and it eliminates downcalls to methods with additional side-effects. Specifically, methods with side-effects cannot make downcalls since they are only allowed to call unoverrideable methods. Also, pure methods cannot have additional side-effects or invalidate the invariant, so it is safe for them to call other overrideable, pure methods. Also, it is safe

1. Guideline L6 also eliminates the need to apply the additional rules given in subsection 5.3.3 for dealing with unoverrideable methods.
2. We could have partitioned the methods into two levels, i.e., overrideable and unoverrideable methods; thus if self-calls to overrideable methods are disallowed, then downcalls would be completely eliminated. However, we wanted to allow downcalls to pure methods since they do not have the reasoning problems related to additional side-effects and the invalidation of type invariants.

```
public class IntCell {

  protected int _val;

  public IntCell(int initVal) {
      set(initVal);
  }
  public void setValue(int newVal) {
      set(newVal);
  }
  public void setFrom(IntCell c) {
      if (_val != c.getValue() ) {
          set(c.getValue());
      }
  }
  public int equals(IntCell c) {
      return _val == c.getValue();
  }
  public /*@ pure @*/ int getValue() {
      return _val;
  }
  protected IntCell(IntCell c) {
      set(c.getValue());
  }
  private void set(int val) {
      _val = val;
  }
}
```

Figure 6.2: `IntCell`'s implementation from the file `IntCell.java`.

for unoverrideable methods to call other unoverrideable methods since such calls do not result in downcalls.

For example, Figure 6.2 is an implementation of `IntCell` that has the suggested calling structure. The private method `set` is called by the overrideable methods `setValue` and `setTo`; therefore, these methods will not be invalidated by subclasses that introduce additional side-effects, such as `CellPlusTotal` and `CellPlusPrevious` from Chapters 1 and 2, because private method `set` cannot be overridden and hence cannot be downcalled. Therefore, the implementation of `CellPlusTotal` given in Figure 1.7 would have been correct in the context of the implementation of `IntCell` given in Figure 6.2.

```
//@ refines "IntCell.jml";

public class IntCell {

  /*@ also
    @ protected code normal_behavior
    @    callable \nothing;     @*/
  public IntCell(int initVal);

  /*@ also
    @ protected code normal_behavior
    @    callable \nothing;     @*/
  public void setValue(int newVal);

  /*@ also
    @ protected code normal_behavior
    @    callable c.getValue();     @*/
  public void setFrom(IntCell c);

  /*@ also
    @ protected code normal_behavior
    @    callable c.getValue();     @*/
  public int equals(IntCell c);

  /*@ also
    @ protected code normal_behavior
    @    callable \nothing;     @*/
  public int getValue();

  /*@ also
    @ protected code normal_behavior
    @    callable c.getValue();     @*/
  protected IntCell(IntCell c);
}
```

Figure 6.3: The code contract for the implementation of `IntCell` given in Figure 6.2.

Figure 6.3 shows the code contract for the methods of `IntCell` based on the code from Figure 6.2; notice that the call of the private method `set` does not appear in the code contract because, as described in subsection 2.6, private methods can be treated as if they were inlined. Furthermore, based on this code contract, the additional side-effects invalidation rule would no longer invalidate method `setFrom` in subclasses like `CellPlusPrevious` and `CellPlusTotal`. Notice also that

methods with side-effects can make object-calls, but they cannot make self-calls to overrideable methods with side-effects.

Furthermore, methods with side-effects should not call overrideable, pure methods because of situations such as the one shown in Figures 2.4 - 2.7. In this example, method `copyFrom` of Figure 2.6 calls down to the pure method `getChange` that expects the subclass invariant involving `_diff` to hold; such calls should not be allowed because, as in this case, the overriding method may access a subclass instance variable constrained by a subclass invariant. Therefore, our guidelines for class libraries do not allow methods with side-effects to call any overrideable methods; this eliminates downcalls and prevents such superclass methods from being invalidated by the Additional Side-Effects Invalidation Rule or the Invariant Invalidation Rule. However, these restrictions also prevent methods with side-effects from calling pure methods even though such calls may not always be unsafe (see the discussion in subsection 6.3.2).

It should also be noted, however, that this calling structure does not eliminate the problems related to aliasing. No special set of guidelines can eliminate the need for the alias control rules either for library implementers or customizers, i.e., the rules from Chapter 3 have to be enforced for both superclasses and subclasses to ensure that pivot fields are declared and that pivot objects are protected. Therefore, the guidelines given in the following subsection include guidelines that state this, i.e., L9 of Figure 6.4 and R3 of Figure 6.5 (see subsection 5.1.2 for the formalization of the alias control rules).

6.1.2 Guidelines for Class Library Implementers

The intent of the guidelines given in Figure 6.4 is to ensure that the implementation of a class library follows the calling structure shown in Figure 6.1 and to make it easier for customizers to extend those library classes. For example, if the implementer of a framework or class library follows these guidelines, then superclass methods cannot be invalidated by new subclasses. That is, these guidelines make sure that methods inherited from the superclass can will always be safely super-called. Furthermore, if the library provider and reuser adhere to these guidelines, then reasoning about creating valid subclasses is simplified.

However, these guidelines are overly restrictive in some cases, but they guarantee that a customizer can easily make a correct subclass without seeing the library's source code. By “easily” we mean that the customizer only needs to follow the simple guidelines for programmers given in Figure 6.5 and described later in subsection 6.1.3; that is, the reuser does not need to read Chapter 2 and does not need to think about the application of most of the rules given in Chapter 2. In particular, these two sets of guidelines guarantee that methods in the library are never invalidated, so super-calls will always be safe.

The first three guidelines, L1 - L3 in Figure 6.4 are necessary for soundness; they are assumptions or are built into the semantics of JML. Guideline L1, in Figure 6.4, is fundamental to behavioral subtyping and is built into JML (see functions *req* and *ens* of the Figure 4.11 and subsection 4.4.4).

For each library class:

L1. Overriding methods should refine the method being overridden.

L2. Methods should not directly assign to instance variables of an object other than the current receiver (`this` in Java and C++).

L3. Methods should not object-call an unoverrideable (e.g. private) method with side-effects.

L4. Methods with side-effects (i.e. non-pure methods) should not self-call overrideable methods.

L5. Unoverrideable methods should not self-call overrideable methods.

L6. Non-private methods with side effects should be overrideable.

L7. Concrete fields should have protected visibility so they are visible to all subclasses.

L8. If a protected concrete instance variable of type T cannot hold all values of type T, then its domain should be described in a public or protected invariant.

L9. Classes and methods should follow the rules for declaring and protecting pivot objects given in Chapter 3.

Figure 6.4: The guidelines for framework and class library designers and implementers.

R1. When overriding a superclass method, always refine it.

R2. Make the subclass's protected invariant imply the superclass's protected invariant, when superclass instance variables have been exposed to the subclass.

R3. Classes and methods should follow the rules for declaring and protecting pivot objects given in Chapter 3.

Figure 6.5: The guidelines for customizers and reusers of a framework or class library that followed the guidelines in Figure 6.4.

Guidelines L2 and L3 are assumptions made in our approach and are necessary for its soundness; these two guidelines, as explained in subsection 2.4.3, prevent problems caused by side-effects to fields of objects other than the receiver. For example, direct assignment to fields of a parameter object could invalidate an unknown invariant, e.g., an invariant of a subtype of the static type of that parameter object; the same problem can also occur unless Guideline L3 is followed.

Guidelines L4 - L8 ensure that customizers do not have to be concerned with the invalidation rules given in Chapter 2. A class is *extensible* if it can be extended, i.e., can have subclasses; a class cannot be extended if it is private or final in the Java sense.

Guidelines L4 and L5 partition the methods of a library class into layers, as shown in Figure 6.1; they allow methods with side-effects to be self-called, but only if they are unoverrideable and thus do not introduce downcalls. Guidelines L4 and L5 also ensure that private methods cannot make downcalls, and therefore, cannot directly or indirectly access or depend on subclass variables. Thus private methods cannot be invalidated by a new subclass.

In summary, Guidelines L2 - L5 ensure that superclass methods can only implement side-effects by directly assigning to instance variables of the current receiver, by self-calling unoverrideable methods, or by object-calling overrideable methods. Therefore, these guidelines prevent inherited superclass methods from being invalidated when a subclass introduces additional side-effects or a subclass invariant. The implementation of `IntCell` shown in Figure 6.2 is an example. Hence, Guidelines L2 - L5 eliminate the need for the Additional Side-Effects and Invariant Invalidation Rules.

Another problem identified in Chapter 2 was unimplementable subclasses caused by unoverrideable methods (see subsections 2.9.2 and 2.9.3). However, if Guideline L6 is followed, then the only unoverrideable methods will be private or (final) pure methods. Therefore, Guideline L6 eliminates unimplementable subclasses when a subclass introduces additional side-effects since the only unoverrideable methods visible to subclasses would be pure methods, i.e., every method that might have to implement additional side-effects is overrideable.

Guideline L7 prevents a subclass method from being unimplementable when a superclass method has been invalidated and cannot be super-called. That is, Guideline L7 ensures that superclass fields can be maintained or accessed even if the superclass methods that access and assigned to those fields have been invalidated and cannot be super-called. Therefore, Guideline L7 eliminates the need for the Mandatory Super-Call Rule given in subsection 2.8.

Guidelines L7 and L8 also prevent the need for concrete data refinement to avoid an unimplementable subclass. Guideline L7 requires that all superclass fields be visible to subclasses, and Guideline L8 requires that the type invariant be visible to all subtypes (both are assumptions in subsection 1.6.6). For example, in JML, the superclass invariant must hold in all visible states of subclasses; this is fundamental to behavioral subtyping and is, therefore, required in JML (see function *inv* of Figure 4.11 and the Invariant Clause Lemma 5.1). However, concrete data refinement would be required if the subclass cannot access or change the necessary superclass variables or the superclass invariant is private. For example, concrete data refinement would be required if the subclass needs to restrict the domain of private superclass fields and the domain, as declared in a private superclass invariant, is not visible. That is, a concrete data refinement would be required unless the subclass can be sure the subclass invariant implies the superclass invariant (i.e., the subclass invariant must be conjoined with the superclass invariant as is done in function *inv* of Figure 4.11). Furthermore, if a

superclass method, that maintains private fields, is invalidated, then the subclass may be unimplementable, unless the subclass implements a concrete data refinement or the superclass code is available.

The guidelines also recommend that methods of library classes not initiate changes to the state of objects except through pivot fields or other owner variables; this prevents the problem of unexpected changes to the abstract value of container objects. That is, Guideline L9 says that library classes should follow the alias control rules given in Chapter 3.

In summary, if all the guidelines in Figures 6.4 and 6.5 are followed, then superclass methods cannot be invalidated by new subclasses and customizers do not have to consider the invalidation rules given in Chapter 2. Therefore, superclass methods can always be safely super-called.

6.1.3 Guidelines for Customizers Inheriting from Libraries

Programmers who use libraries or frameworks that follow the above guidelines, but are not themselves producing extensible libraries or frameworks, do not need to follow the guidelines given in Figure 6.4. However, customizers need to pay attention to the guidelines for reusers given in Figure 6.5.

In particular, if implementers of the inherited library classes have adhered to the guidelines in Figure 6.4 and the customize or follows the guidelines in Figure 6.5, then super-calls will always be safe. That is, if the calling structure is as shown in Figure 6.1 and there are no recursive methods, no non-refining methods, and the superclass invariant holds, then super-calls will always be safe. Note that the semantics of JML requires that Guidelines R1 and R2 be followed.

Also, Guideline R3 recommends that customizers not change the state of objects returned from method calls, i.e., from methods of objects in library classes. Thus methods of library classes would not have to be inefficient and make deep copies of pivot objects to protect them from clients and customizers. Specifically, Guideline R3 says that clients and customizers should follow the rules given in Chapter 3.

6.2 Tool Support

Another important way to make our technique less of a burden for class library implementers and customizers is through tool support. Tool support is necessary to automatically create code contracts and to assist in enforcing the rules. This section briefly describes the tool that would fill this role for JML and Java.

Our proposed tool will have two phases after the initial Java type checking. It will work on a per-class basis, looking at related classes mentioned in the code. For a given class, the first phase will generate its code contract (i.e., the callable clause and possibly the assignable clause). Generating the assignable and callable clauses can be done as demonstrated in the T-rules of Figures 5.1 - 5.3, i.e., instead of checking whether or not the assignment or method call is allowed, the tool would add that field or method to the assignable or callable clause respectively.

The code contract always reflects the current calling structure of a class no matter how many layers there are in the class hierarchy; that is, the code contract of a method defined in the subclass is derived directly from the code of that method, but the code contracts of inherited methods are inherited in the same way that methods are inherited by subclasses.

Phase 2 will check to make sure, based on the T-rules given in Figures 5.1 - 5.3 and the predicates in Figures 5.4 - 5.8, that all methods have been properly overridden and that no invalid super-calls have been made. For example, the tool may say that a superclass method should be overridden, or it may complain that an overriding method made a super-call to an invalidated superclass method. The checking done in Phase 2 would be an implementation of the T-rules given in Chapter 5 (note, however, that the predicates given in Figure 5.8 would supersede those given earlier in the chapter to make our technique less restrictive and more practical).

The T-rules also enforce our alias control rules from Chapter 3. Thus the checker would also issue errors when an improper assignment to a pivot field is detected (i.e., the right hand side is not a new object constructor call or null), or when an actual parameter to a method call is not an owner variable (i.e., the called method has permission to assign to fields of that argument object).

6.3 Discussion

6.3.1 Recursive Methods

Customizers also have to be concerned about recursive methods and termination when superclass code is unavailable. Guideline L10, when combined with our alias control rules, prevents mutual recursion and callbacks among methods of unrelated classes and objects[3]. Furthermore, Guidelines L10 and R4 eliminate mutual recursion between superclass and subclass methods, so they eliminate the need for customizers to think about the Callback Cycle Invalidation Rule. Note, however, that Guideline L10 does not prevent unoverrideable (private or final pure) methods from being recursive or mutually recursive with other unoverrideable methods since these methods cannot make downcalls when Guideline L5 is followed.

L10. Overrideable methods should not be recursive or mutually-recursive with other methods.

Guideline R4 for customizers is also helpful for reasoning about termination of recursive methods. The main principle of these two guidelines is to ensure that customizers and verifiers can reason about method termination by ensuring that the code of all methods that could be executed are available to verifiers.

---

3. Since there are patterns involving callbacks, libraries could make the code of methods available to reusers if they are involved in mutual recursion among unrelated classes; this would allow reusers to reason about termination.

R4. Avoid creating a group of mutually recursive methods involving a method of a library superclass.

6.3.2 Another Guideline for Libraries

There are situations where the guidelines in Figure 6.4 are more conservative than necessary. For example, we could allow methods with side-effects to call pure overrideable methods under some circumstances (even though this violates Guideline L4 and our rules). That is, the problem caused by subclass invariants, as illustrated by class `CellPlusInvariant` in Figures 2.4 - 2.7, can be avoided by requiring that all calls to pure overrideable methods be made prior to any changes in the state of the receiver object. If this is possible, then the subclass invariant will always hold when the call (or downcall) is made. The following guideline expresses this idea.

L11. Methods with side-effects should only call overrideable pure methods when the pre-state of the receiver object has not been changed.

Note, however, that we do not include this guideline in our recommendations because it does not follow the rules given in Chapter 2. For example, if the guidelines in Figure 6.4 are followed by the superclass and the guidelines given in Figure 6.5 are followed by the subclass, then none of the superclass methods can be invalidated by any of the rules given in Chapter 2. However, this would not be true if we included Guideline L11, i.e., such methods could be invalidated by the Invariant Invalidation Rule (see subsection 2.4.1). Furthermore, our formalization has no way of checking that Guideline L11 is being followed. Also, since Guidelines L4 and L11 are incompatible, L4 would have to be modified to accommodate L11.

6.3.3 Informal Documentation

What do these guidelines say about informal documentation for class libraries and frameworks?

One clear conclusion is that the notion of documentation as a contract [LG86, Mey88, Mey92] is essential. For example, Guideline L8 says that libraries have to be documented with public and protected invariants, and this is required by Guideline R2 as well. Such contracts benefit greatly from formality, but even informal contracts are better than none. Informal contracts can be structured into invariants, pre- and postconditions for methods, and assignable clauses to help make them more understandable and readable [LG86]. We believe that the division of specifications into public and protected parts is another way to help make such specifications more understandable, since it separates the information needed by clients from that needed by customizers writing subclasses.

Guidelines L4 and L5 divide the methods of a library class into layers, as illustrated in Figure 6.1. The layers themselves, along with other guidelines, are a substitute for the code contract. That is, they eliminate the need for library documentation to include something like the code contract. However, if a

library provider cannot strictly follow this layering, something like the callable clause in the code contract needs to be part of the documentation provided for the library. Since these are essentially lists, there seems to be no reason not to use the formal notation and automatic tool support for generating them.

Finally, documentation for a library needs to discuss the guidelines for reusers, and especially to highlight Guideline R4 so reusers understand why and how to avoid mutually-recursive methods when super-class code is unavailable. This documentation should also highlight Guideline R3 and specify that (or at least when) it is unsafe for customizers or clients to modify the state of objects returned from methods of library classes. Otherwise, library class implementers and customizers should never expose pivot objects to clients.

6.3.4 Conclusions

Following both sets of guidelines from Figures 6.4 and 6.5 ensures that subclasses do not invalidate superclass methods; this means that methods can be implemented and reasoned about independently from the implementation of superclass methods. Our guidelines could also be used to help with the organization of a class library or framework, i.e., its implementation could be reviewed for its reusability using the code contracts of its classes in light of the guidelines for library providers shown in Figure 6.4. These two sets of guidelines also give insight into the information that needs to be included in documentation for library classes and frameworks.

Guidelines L2, L3, L5, L6, and L7 hold automatically in Smalltalk, i.e., only instance variables of the current receiver are in scope and all methods are overrideable. Furthermore, Guidelines L1 and L8 are necessary for behavioral subtyping and for method verification, but are not part of the T-rules or static checker. Therefore, only Guidelines L4 and L9 would have to be enforced by a checker for Smalltalk programs. Since large systems have been coded in Smalltalk and because so many of the guidelines are automatically enforced by Smalltalk itself, this provides good evidence that the restrictions imposed by these guidelines are not going to be impractical for library implementations.

CHAPTER 7: CONCLUSION

7.1 Summary

In Chapter 2, we gave a set of rules for preventing the invocation of superclass methods when those methods may no longer satisfy their superclass specification or may have unverifiable side-effects; we also illustrated how and why these problems arise when superclass methods make downcalls. Our rules prevent the problems caused by downcalls by disallowing super-calls and by requiring method overrides. That is, we provided a set of invalidation rules to prevent super-calls to methods that may have unverifiable behavior and we provided an analogous set of overriding rules to prevent self-calls and object-calls to invalidated superclass methods.

The soundness of our reasoning technique requires that there be no unexpected changes in the abstract value of an object. Therefore, in Chapter 3, we gave another set of rules for ensuring that pivot fields are declared and that pivot objects are protected from unexpected side-effects. We gave examples to illustrate why declaring and protecting pivot objects is necessary for the soundness of any verification logic and for the soundness of the assignable and represents clauses used by our technique.

To prove that the rules given in Chapters 2 and 3 prevent the problems described, we needed to show, based on our rules, that there is a sound, modular proof system for verifying the correctness of subclasses without superclass code. In Chapter 4, we formally defined the syntax of Java-C, a subset of Java, and JML-C, a subset of JML; these languages only include the features needed to demonstrate that our technique is sound and sufficient for preventing the unverifiable behavior and unexpected side-effects described in Chapters 2 and 3. We also specified an operational semantics for Java-C and an axiomatic semantics for JML-C. As required, the operational semantics includes rules and data structures to represent and handle objects, dynamic binding, and the run-time stack and heap. In Chapter 4, we also provided some examples to illustrate how the axioms and inference rules of the verification logic would be used to verify correctness of subclass methods using JML-C specifications; the verification was done without the need for superclass code.

In Chapter 5, we formalized the rules from Chapters 2 and 3 as a statically enforceable type system, i.e., as the set of T-rules. The T-rules were then used in the soundness proof of the verification logic from Chapter 4. The soundness of this verification logic proves that our technique permits sound, modular verification of subclasses without superclass code.

In Chapter 6, we investigated ways in which the implementers of a class library can make their frameworks more user friendly. That is, we provided a set of guidelines that, if followed, make it easier for customizers to extend the classes in these libraries. These guidelines ensure that super-calls are always safe and that our verification logic can be used to prove the correctness of subclass methods without superclass code.

In the sections that follow, we will discuss related work and some of the results and contributions of our research. In Section 7.2, we describe research related to our technique for handling downcalls. In Section 7.3 we describe some of the work related to object invariants. In Section 7.4 we discuss research related to aliasing, and finally, in Section 7.5, we describe some conclusions and directions for future work.

7.2 Research Related to Correct Subclassing

In this subsection, we discuss papers related to our technique for preventing problems caused by downcalls. This subsection is adapted from our OOPSLA paper [RL00].

In a paper presented at OOPSLA'98, Leino introduced the notion of data groups and dependencies for controlling which subclass fields can be modified by an overriding subclass method [Lei95, Lei98]. The maps and in clauses in JML are derived from Leino's work [LPHZ02]. A specification language needs a feature like these data group clauses to declare dependencies so a tool can apply the Additional Side-Effects Invalidation and Overriding Rules. However, Leino's work does not attempt to solve downcall problems caused by subclassing.

Kiczales and Lamping informally describe the kind of documentation that needs to be provided by an "extensible class library"[KL92]. Kiczales and Lamping show that more knowledge of the calling relationships among methods is needed by programmers inheriting from a class library. They propose that methods be organized into layers; a method may call another method, only if the other method is a member of the same layer or is on a lower layer [KL92][section 4.8]. This is similar to our guidelines for library providers that organize the methods of a class into three levels. However, in our guidelines, the public non-pure methods are not allowed to call each other, which is key to preventing invalidation of superclass methods. Kiczales and Lamping also propose that methods be grouped based on the instance variables manipulated by methods within the group; every method in such a group would be overridden whenever any member of the group is overridden [KL92][sections 4.5 and 4.6]. Thus, a group would have to be inherited as a whole by subclasses. The documentation they propose is informal, and thus, does not allow static checks for possible problems when new subclasses are created.

Lamping later formalizes some of these ideas into a type system approach for describing what he calls the *specialization interface*, an interface between a class and its subclasses [Lam93]. An important benefit of this technique is that it allows for additional error detection when new subclasses are created. However, in contrast to our technique, Lamping's does not say anything about super-calls and requires that entire groups be overridden. Our technique only requires that methods be overridden if they have been invalidated, i.e., if they have additional side-effects or need to establish a subclass invariant. Also, Lamping's work [Lam93, KL92] does not attempt to solve the problems caused when the superclass code is unavailable to customizers.

Steyaert, Lucas, *et al.* introduce a similar approach called "reuse contracts" for specifying a contract between a class and its subclasses [Luc97, SLMD96]. Like a specialization interface, a *reuse contract* specifies the calling interdependencies of methods of a class, that is, a reuse contract lists the other methods on which a particular method depends. The primary innovation of this approach is in defining a set of operators on reuse contracts that allow safe transformations to the calling structure. It also allows the detection of conflicts between a class and its subclasses due to changes in the calling structure of the superclass, and it formalizes the meaning of correctly implementing a reuse contract. However, a reuse contract does not necessarily list all methods called. Only those methods manually determined to be important for inheritors are included, and no guidelines are given for how to do this. In addition, reuse contracts are primarily concerned with the evolution of superclasses, while our work is concerned with the addition of new subclasses and their effect on the behavior of superclasses.

All of the above approaches are syntactically based, that is, they do not necessarily detect or prevent errors caused by changes in the behavior of methods overridden by subclasses.

Perry and Kaiser [PK90] address the semantic problem caused by changes in the behavior of methods overridden by subclasses in the context of their work on testing. They point out that inherited superclass methods must be retested unless "the new subclass is a pure extension of the superclass, that is, … there are no interactions in either direction between the new [subclass] instance variables and methods and any inherited instance variables and methods." They further show that a different set of tests may be needed to retest these inherited methods. However, our technique does not limit the interactions between superclasses and subclasses so severely. Further, if the documentation and reasoning technique we propose is followed, then inherited methods would not need to be retested or reverified.

Stata and Guttag [SG95] solve this semantic problem by requiring that new subclasses implement behavioral subtypes [Ame91, DL96, LW93, LW94] of their superclass. They extend the partitioning ideas of Kiczales and Lamping [KL92] and Lamping [Lam93] into a formal system of class components composed of disjoint sets of methods and instance variables. No method in one component is allowed to directly access variables in another component, and all methods within a component must be overridden whenever any one of the methods in the component is overridden. This permits individual components to be implemented, reasoned about, and overridden independently of other components of a class. However, in this formalization, improvements to individual methods cannot be made without overriding all methods of a component, even when the modifications would not change observable behavior. Edwards weakens this requirement by allowing individual methods to be overridden as long as the representation invariant of instance variables of the component is maintained [Edw96]. Neither Stata and Guttag nor Edwards give conditions under which super-calls may be made.

Like Stata and Guttag, JML requires that subclasses implement behavioral subtypes. However, like Edwards, JML also allows improvements to individual methods by specifying the representation

invariant (i.e., the protected invariant) in the protected specification. JML's specification inheritance ensures that the subclass representation invariant implies the superclass representation invariant; thus, in JML, an individual method may be overridden as long as it refines the method it overrides. Furthermore, although JML does not allow non-refining methods or all forms of concrete data refinement, we do provide rules for reasoning about how to safely create subclasses with such methods and data structure changes.

Stata later separates the notions of subtyping and subclassing, as we do, to allow overriding parts of a component [Sta97]. Overriding parts of a component is permitted if the superclass representation invariant is maintained by the new subclass (as Edwards requires). Stata also proposes conditions to allow super-calls. Super-calls are allowed if the specification of each overridden superclass method is refined by the specification of the new subclass method. This condition, like the method refinement rule, only ensures that superclass method invocations satisfy the superclass method specification. When superclass code is not available, this condition does not handle verification problems caused by additional side-effects or subclass invariants, i.e., superclass code would have to be available and the behavior of super-calls reverified. Our approach, however, handles these problems and sometimes requires that fewer methods be overridden by providing the calling structure of the methods in a class and rules for determining which methods to override. Also, there is no need to explicitly partition methods and instance variables into components, although our tool could be used as an aid in creating and enforcing such a partitioning.

Mezini proposes a metalevel *cooperation contract* that allows library designers to declare properties of classes that are propagated to subclasses [Mez97]. These properties are specified in a cooperation contract language (CCL). The cooperation contract allows base classes to be monitored to detect modifications to a superclass that may invalidate existing subclasses. Mezini incorporates ideas from Lamping [Lam93], Stata and Guttag [SG95] and Steyaert, Lucas, *et al.* [SLMD96]. For example, class designers can partition methods or classes into groups, can express dependencies such as when certain methods must be called, or can specify when methods are required or are non-overrideable. Although super-calls are shown in examples, it is unclear whether the mechanism ensures their safety and no claim is made as to how to reason about such super-calls. Cooperation contracts are entirely syntactic, so they do not contain enough information to prove correctness of a new subclass. In addition, this method cannot be used for languages, such as C++, Java, Smalltalk and Eiffel, unless extended to have a *metaobject protocol* [KdRB91], whereas code contracts can be generated and used by any statically typed, OO language.

The approaches described above would be carried out as part of the analysis and design activities for a class library; furthermore, the determination of what information is included is done manually, whereas the code contract, a major part of our approach, would be generated automatically by our proposed tool. A tool that ensures that no rules are violated eases the work of applying the rules and thus automates some of the work involved in creating correct subclasses. In addition, except for Stata

[Sta97], the above approaches do not handle downcalls caused by super-calls other than ignoring or prohibiting them. However, even Stata's work does not handle reasoning about additional side-effects or subclass invariants when superclass code is unavailable.

Szyperski shows how downcall and callback problems are avoided by using object composition and message forwarding rather than implementation inheritance [SGM02][pp.133-135]. Object composition means building an object from other objects. The contained objects perform tasks for the containing object. *Forwarding* means sending a message on from one object to another object. Szyperski's technique simulates implementation inheritance by forwarding method calls to a contained object; this contained object would have the type of what would have been the superclass. Forwarded calls are somewhat like super-calls except that forwarded calls do not create downcall problems because, once control has been passed to the contained object, control stays inside its methods unless the contained object itself has a reference to the original object. Thus, even though object composition eliminates many downcall problems, it does not address the problems related to aliasing and unexpected side-effects. Furthermore, object composition works fairly well in the design of some classes, but protected methods will not be available to subclasses when implementing new subclass methods. Furthermore, Szyperski gives an example showing that implementation inheritance cannot be simulated in all cases by object composition and method forwarding, and other examples are difficult and complicated [SGM02][p. 135-138].

Our study focused on the semantic fragile subclassing problem, that is, how to create valid subclasses using only specifications. This problem is closely related to the semantic fragile base class problem because both problems are caused by downcalls and changes to the calling structure of classes. Mikhajlov and Sekerinski describe the fragile base class problem and give four requirements for disciplining inheritance to avoid such problems [MS98]. Their requirements prohibit access to superclass instance variables and do not allow instance variables to be declared in subclasses. Our technique, however, allows both, i.e., declaration of subclass variables and access to superclass instance variables by subclass methods and invariants. The method specifications used by Mikhajlov and Sekerinski implicitly document what methods may be called. This information is used to disallow overriding a method in a callback cycle, since this would introduce a “new cyclic method dependency.” But, our technique, which relies on similar information in the callable clause, allows methods in such cycles to be safely overridden. Furthermore, the callable clause is automatically generated. Their technique, like ours, requires that superclass method specifications be used when verifying subclass methods. However, their technique prevents the problems caused by additional side-effects and subclass invariants by placing severe restrictions on subclasses, i.e., by disallowing declaration of subclass fields or access to superclass fields by subclasses. In contrast, our rules allow sound reasoning about the state of subclass instance variables, additional side-effects, and subclass invariants without requiring these restrictions.

Changes in the code contract could also be used to catch problems such as method capture. *Method capture* occurs when a new method is added to a (super) class in a new version of the library and that method was already defined in an existing subclass [Luc97, SLMD96]. Therefore, the subclass would have to delete or rename the captured subclass method or make sure the captured method refines the method it now overrides.

To summarize, our approach has important advantages over all the work described above in that it allows super-calls and reasoning about the safety of such calls. Furthermore, creation of the code contract need not be part of the manual design activities of a class library.

7.3 Research Related to Object Invariants

Liskov and Wing [LW94] require that the subclass invariant imply the superclass invariant, as we do, but this requirement is not sufficient for modular soundness. That is, reasoning about type invariants that depend on the state of internal objects is not sound without some form of alias control.

Huizing and Kuiper [HK00] present a verification logic for an object-oriented language with class invariants. As in Liskov and Wing above, they require that the subclass invariant imply the superclass invariant but, without proper restrictions on aliasing, this technique is also unsound for layered object structures; also, Huizing and Kuiper do not say how their technique would ensure that the subclass invariant holds prior to a downcall. Thus superclass methods would have to be reverified in the context of the whole program, including every new subclass, but this is not modular and requires superclass code.

Barnett *et al.* [BDF+04] developed a way of statically determining when type invariants are guaranteed to hold. Two auxiliary variables are defined for each object. One variable indicates the type lowest in the hierarchy for which the invariant holds; thus, using our notation, this special variable equals T when *inv*(T) holds for the object (i.e., the invariant of subclasses of T may be invalid). The second variable has type boolean and indicates when the run-time type invariant holds for the object. The validity of the invariant is then tracked for each statement in the program. These auxiliary variables are updated by two special statements to indicate when parts of the invariant are no longer valid and when they hold again. Aliasing is unrestricted but, to prevent unexpected side-effects, changes to the state of an object have to be initiated by an owner. A disadvantage of this technique is that method specifications can become cluttered quite quickly because pre- and postconditions have to include assertions about these special variables for each argument object; these assertions are needed in method specifications so the static checking can be done modularly. In contrast, our technique automatically ensures that the run-time type invariant holds when necessary. As future work, however, we would like to loosen our restrictions on actual parameters[1]. At the same time, our goal is not to require any annotations other than method specifications (requires, ensures, assignable, and callable clauses) and data specifications (in, maps, and represents clauses); so far we have been able to do this. Barnett *et al.* do not handle invariants over cyclic data structures, nor does our technique.

Leino and Mueller [LM04] extend Barnett *et al.* but they use the more flexible Universe type system [Mül02, MPH01] to control aliasing and side-effects. The subclass type invariant is allowed to depend on superclass fields, fields transitively owned by the subclass, and fields reachable from an internal object. Thus invariants are allowed to depend on cyclic data structures. They also allow ownership transfer. Our technique is not as flexible, i.e., the syntax of invariant clauses is more restricted, we do not allow ownership transfer, and we restrict actual parameters in method calls to owner variables (to ensure that the invariant holds for argument objects). On the other hand, our technique is fairly simple compared to their technique and it seems unclear when and how to properly use the two new statements used in Barnett *et al.*

Mueller, Poetzsch-Heffter, and Leavens [MPHL06] have also developed a technique for reasoning about invariants over layered object structures. That is, they allow invariants to depend on the state of internal objects and cyclic data structures; for soundness, alias control is enforced by the Universe type system [Mül02, MPH01]. They also use visibility-based invariants, i.e., declarations of an invariant must be visible in every method that might invalidate that invariant. Visibility-based invariants permit additional dependencies across underlying layers of objects. However, this does not allow subclass invariants to depend on superclass fields, whereas we do allow such dependencies. An area of future work would be to investigate the possible integration of visibility-based invariants into our technique to loosen our restrictions on invariants and actual parameters.

Barnett and Naumann [BN05] also extend Barnett *et al.* with visibility-based invariants but they do not handle cyclic data structures.

7.4 Research Related to Aliasing

The approaches to dealing with aliasing can be divided into four broad categories: detection, advertisement, prevention, and control [HLW+92]. *Alias detection* means determining either statically or dynamically where the potential and actual aliases occur in a program. *Alias advertisement* means declaring the aliasing properties of methods so detection of unwanted aliasing can be done modularly. *Alias prevention* means adding constructs to the programming or specification language so a program analyzer can statically check and guarantee there is no aliasing within a particular context. *Alias control* means isolating the effects of aliasing in the run-time state.

Alias detection appears to be impractical because the static interprocedural analysis necessary to detect it is NP-hard [LR91]; thus it is likely to be too slow. Programmers could write code to detect aliasing at runtime so they can take evasive action, but this is not always possible because of limitations in programming languages [HLW+92].

1. Another possibility would be to have invariants that always hold, e.g., an invariant that depends on some boolean model field [LM04, Lea06]. Specifically, the invariant would be implied by some field that specifies whether the consequent of the implication holds; with such invariants, read-only variables could be allowed as actual parameters since the invariant would always be true and it would be possible for the verifier and implementer to use this information in reasoning about correctness.

Alias control[2] is based on the analysis of state reachability, i.e., determining whether the system will ever reach a state in which there is unexpected aliasing. Aliasing control is applied when the effects of aliasing can only be determined by taking into account the run-time state of the system [HLW+92]. The object-oriented language SPOOL [AdB90] uses a system of alias control as the basis of its proof system. Some effect systems [LG88, GB99] control aliasing and unexpected side-effects through regions to restrict how the state of the system can be accessed. Wills [Wil91] bases alias control on *demesnes*, a set of objects that participate in the representation of an abstraction.

Our technique falls into the category of alias prevention, i.e., we prevent objects from being referenced by more than one owner variable in the same context, but otherwise we do not restrict aliasing. Furthermore, we require that changes to the state of objects be initiated through an owner variable; we use the assignable clause to modularly enforce this restriction on side-effects.

This section is organized a little differently because there is a large amount of research on aliasing. In subsections 7.4.1 and 7.4.2 we review some of the techniques that fall into the categories of alias prevention and alias advertising. In the last subsection 7.4.4, we compare and draw conclusions.

7.4.1 Alias Prevention

In this subsection, we review of some of the alias prevention techniques for dealing with aliasing and preventing unexpected side-effects due to representation exposure.

7.4.1.1 Islands

Hogg [Hog91] proposes that the existence of aliases be rigorously controlled through "islands of aliasing". An *island* is the set of internal objects accessible from a bridge object. An island guarantees that no object other than the bridge or members of the island hold a reference to any object internal to the island. That is, no access paths to internal objects are allowed to exist except through the bridge object.

Islands are implemented through access modes, i.e., each variable has a declared access mode. There are four *access modes*: unrestricted, read, unique, and free. The *read mode* specifies that the variable has read-only access to the object it references. The *unique mode* specifies that the variable is the only one in the system that references a particular object. The *free mode* specifies that no variable in the system references a specific object.

Clearly a variable cannot be free without a special way of reading that variable. That is, Islands need a new atomic operation, the destructive read, for handling unique and free variables. The *destructive read operation* returns the value held in a variable (i.e., a reference to an object) and sets the variable to null. If applied to a unique variable, this operation, in effect, deletes the only reference to the object and thus converts a unique expression into a free expression. Furthermore, a free variable

2. We have been using this term in a more generic sense, i.e., we have been using alias control to mean any technique that prevents unexpected side-effects due to aliasing. After this paragraph, we will return to the more generic meaning.

may only be accessed via a destructive read to ensure that the object reference remains free. A new object constructor call also returns a free object reference.

There is a set of rules for ensuring that the aliasing properties of an island are maintained. For example, a read variable may not be the left side of an assignment statement, and unique variables may only be assigned the result of free expressions. Also, because the destructive read has side effects, it cannot be applied to a read variable.

7.4.1.2 Balloons

Almeida [Alm97] proposes Balloon Types as another method of controlling aliasing. The type system enforces restrictions on the code permitted in methods of a Balloon class to ensure that none of its internal objects are statically aliased by any external object. Dynamic aliasing of internal objects of a Balloon is allowed.

Almeida gives the following invariant that must be maintained for any balloon object B:

(1) There is at most one static access path to B in the system.

(2) Any reference to B must be from an object external to B.

(3) No object external to B may reference any object internal to B.

This invariant is enforced for Balloon Types by a set of rules. For example, (1) and (2) are enforced through restrictions on assignments, i.e., a reference to an already existing Balloon object cannot be assigned to any variable. This is the only restriction that also applies to methods of non-balloon classes. Thus no methods are allowed to include assignment statements that create a static alias to an existing Balloon object.

7.4.1.3 Unique Variables

Baker [Bak95] proposes that programming languages include “use-once” variables in addition to the usual variables. A *use-once variable* is assigned a value exactly once and subsequently read exactly once. The read of a use-once variable would have to be a destructive read that nullifies that variable immediately afterward. Thus the use of a variable in one expression disallows its use in any other expression; this property can be checked statically. Objects referenced by a use-once variable have the property that there is only one access path through which they can be referenced at any given time.

Minsky [Min96] argues that aliasing caused by pointers (object references) can be avoided through the similar concepts of “unique pointers” and unshareable objects, also called *u-objects*. Variables that hold the only reference to an object are called *u-variables*. Minsky extends the Eiffel programming language to incorporate support for u-variables and u-objects.

U-variables are similar to the unique variables in Hogg’s Islands and are handled through a similar set of type rules. This technique also requires a special move-assignment statement analogous to the destructive read operator. A *move-assignment* statement copies the object reference from the source variable into the target variable and then assigns null to the source. The compiler automatically

interprets an assignment statement in this way when the right side is a u-variable. Also, regular variables may not be assigned to u-variables.

7.4.1.4 Unique Variables without destructive reads

Boyland [Boy01] demonstrates that the uniqueness property of a variable can be preserved with existing language features, e.g., without the need for destructive read operators. He describes a technique for annotating the program in such a way that a static checker can ensure that the uniqueness property is preserved. The basic idea is that a unique variable can only be read after any previously existing aliases have been eliminated. That is, a unique variable can temporarily be aliased, but it cannot be read until the uniqueness property has been reestablished.

7.4.2 Alias Advertisement

In this subsection, we review some of the alias advertisement approaches to preventing the kind of aliasing and representation exposure that could lead to unexpected side-effects.

7.4.2.1 Ownership Types

Ownership types as embodied in the Flexible Alias Protection system [NVP98, CPN98, CNP01] is an example of alias advertisement. It uses aliasing modes to advertise which objects can be exported and which must be encapsulated. It does not require that all internal objects be unaliased via external objects; thus it is not an alias prevention technique. It has a formal type system to enforce the rules of the technique [CPN98].

Flexible Alias Protection is a programming discipline that allows the designer of an abstract data type to hide information about the internal representation of the abstraction while allowing static aliases to other internal "argument" objects. To protect the abstraction from unexpected changes, the discipline restricts the way argument objects are used.

The internal objects of a class are divided into several groups: representation, argument, free, value, and mutable objects. *Representation objects* can be modified, but only by methods of the object to which they belong. Representation objects can be stored in and retrieved from internal objects, but cannot be exposed (exported or aliased) outside the abstraction. *Argument objects* can be aliased, but the abstraction cannot modify or depend on the state of these objects; furthermore, methods of argument objects cannot be called if they access any mutable state [NVP98][p. 171]. *Free objects* are not referenced by any variables in the system (as in Islands). *Value objects* are immutable and therefore can be exported and aliased. *Mutable objects* can be modified, exported, and aliased.

Clarke, Potter, and Noble [CPN98] formalize a type system for a slightly simplified version of Flexible Aliasing and give a proof of its soundness. Clarke and Drossopoulou [CD02] extend the Flexible Aliasing discipline and the notion of ownership types with a proof system for reasoning about the absence of aliasing; they also include an effects system for reasoning about the non-interference of computation, i.e., the absence of unexpected side-effects. For example, when two types are disjoint, then fields having these types cannot be aliases.

7.4.2.2 Universe Type System

The Universe Type System [MPH01, Mül01, Mül02, DM05] is a technique to statically control aliasing; it primarily builds on the ideas from ownership types. The Universe Type System enforces a hierarchical partitioning of the object store into universes and controls references between universes. However, to increase flexibility, it includes read-only references that can be exported across universe boundaries, i.e., outside an abstraction. Fields, local variables, and parameters must be annotated to indicate whether the reference is read-write (part of the owner object's representation) or read-only (could be part of the representation of a different object).

Static and dynamic aliases are controlled by ensuring that the following invariant holds for a program: If there is a reference from object *X* to object *Y*, then at least one of the following must hold:

1. *X* and *Y* belong to the same universe,
2. *Y* belongs to a universe owned by *X*, or
3. the reference is read-only.

This invariant guarantees that a modification of an object's abstract representation is only possible by calling a method on an appropriate owner object. An advantage of the Universe Type System over Flexible Alias Protection is that representation objects can be exported and aliased across universe boundaries via read-only references, i.e., they can safely be exposed outside the abstraction.

Allowing read-only references to migrate outside of an abstraction results in the same problem that we faced with invariants. That is, the run-time type invariant of an object referenced by a read-only reference is not guaranteed to hold without additional restrictions. Mueller teamed with others to present several possible solutions to this problem (see Section 7.3). We temporarily solved the problem by not allowing non-owner variables as the actual parameter in method calls. As future work, we hope to loosen this restriction.

7.4.2.3 Pivot fields

Detlefs, Leino, and Nelson [DLN98] describe a specification technique for preventing representation exposure in many situations through the declaration and encapsulation of pivot objects. Leino and Stata [LS99] extend these ideas with a specification technique for preventing some additional forms of unwanted aliasing in programs built in layers of abstraction. Their focus is to protect the pivot objects of a class from unwanted aliasing. Pivot fields are determined from the depends clauses [Lei95, Lei98] appearing in class declarations. Pivot fields and pivot objects have the same meaning here and in our technique[3].

Pivot objects are similar to the representation objects of Flexible Alias Protection and the Universe Type System in that they contain data related to the value of the higher-level abstraction. Therefore, to avoid conflicting or unexpected changes to the state of the lower-level abstraction, mutable pivot

3. The depends clause is a precursor of the in and maps clauses from Leino *et al*. [LPHZ02]. JML used the depends clause before changing to the in and maps clauses.

objects are not allowed to be aliased. To guarantee that pivot fields are not aliased, the specification must be strong enough so that a pivot field:

1. will not be assigned a reference to an input argument unless the argument is unaliased,
2. will not be assigned the result of a method call unless this result is unaliased, and
3. can safely be passed as a parameter to a method or constructor without being statically aliased.

Leino and Stata solve this problem based on the notion of virginity for objects. They add three boolean-valued auxiliary variables to every object: *virgin***,** *pivot***,** and *plenary.* The program is not allowed to explicitly modify these variables. An object is *virgin* if it is not, and never has been, reachable from a global variable. This auxiliary variable is true when the object is first created and then is automatically set to false as soon as a reference to the object is assigned to an instance or global variable. The auxiliary variable *pivot* indicates whether or not the object is referenced by a pivot field, and *plenary* indicates whether or not the object is referenced by a non-pivot location. A *non-pivot location* is a global variable or non-pivot field. This approach uses the depends clause that specify dependencies like those described previously for JML specifications using the in and maps clauses.

The techniques discussed in previous sections are based on type systems. However, the current technique is based on a specification technique and a set of restrictions that can be statically checked modularly. To be checked modularly, assertions about these auxiliary variables have to be included in method specifications. Also, leaking of pivot objects from a context is prevented by disallowing the return of a pivot object from a method call.

7.4.2.4 Side-effects, data groups, and pivot fields

Since the soundness of our rules from Chapter 2 depends on the soundness of the assignable clause, we needed some way of ensuring that there could be no unexpected side-effects. Leino *et al.* [LPHZ02] describe a sound, modular technique for specifying and statically checking side-effects in methods of an object-oriented programming language. This technique was appealing because, syntactically, it only required data groups and a clause for specifying side-effects, both of which were already included in JML.

Our technique is an extension of this work by Leino *et al*. For example, our technique, like theirs, requires that a pivot field be the first to contain a reference to a newly created object; also, assignment to a pivot field that creates an alias is not be allowed. However, to prevent unexpected side-effects to pivot objects, Leino *et al.* also enforce other restrictions to ensure that pivot fields are unique. Our rules in Chapter 3 do not enforce this restriction, i.e., in our technique, pivot fields are allowed to be aliased by non-pivot variables. Thus our other rules are less restrictive.

To allow aliasing of pivot objects and avoid unexpected side-effects, all non-owner variables are automatically read-only in our technique. However, as mentioned above, read-only variables may reference objects with an invalid invariant, so we had to restrict actual parameters in method calls to achieve soundness. Also, as mentioned earlier, we plan to loosen these restrictions in future work.

We also replaced the depends clause [Lei95, Lei98] in JML with the in and maps clauses from Leino *et al.* [LPHZ02] because the in and maps clauses automatically enforce the visibility requirements of data dependency declarations (subsection 2.2.1.2).

Finally, our technique, unlike Leino *et al.*, deals with correctness of method implementations beyond side-effects; that is, we handle, pre- and postconditions, invariants, subclassing, and downcalls in our verification logic.

7.4.3 Controlling Side-Effects

We needed the rules in Chapter 3 to prevent assignment to fields not allowed by the assignable clause; this is also necessary to prevent the abstract value of an object from changing unexpectedly. Leino and Nelson [Lei95, Lei98, LN02] were the first to use dependencies and public data groups to control assignment to concrete fields hidden from clients. Mueller *et al.* [MPHL03] generalized the modularity rules from Leino and Nelson's work to handle dynamic dependencies, i.e., to handle pivot objects. Pivot objects determine the abstract value of the enclosing object so the soundness of a verification logic depends on preventing unexpected changes to these internal objects.

As mentioned in the previous subsection 7.4.2.4, our technique is adapted from Leino *et al.* [LPHZ02]; however, instead of encapsulating object references in unique variables as they do, we encapsulate side-effects without restricting aliasing. That is, we require that side-effects be initiated through owner variables so changes to the state of an internal pivot object will be encapsulated in (methods of) the object containing the pivot field. Also, in our technique, an object can have only one owner variable that is visible in a given context (i.e., formal parameters are temporary owners and cannot be aliases of other owner variables visible in the same context).

Lu and Potter [LP06] and Skoglund [Sko02] have techniques that also focus on encapsulating side-effects rather than object references; however, these techniques require special annotations in all variable declarations so their restrictions can be modularly enforced by a type system. Therefore, these techniques have a large syntactic overhead compared to ours, i.e., we do not require that variable declarations include special annotations other than the data group clauses needed for specifying dependencies and controlling side-effects.

7.4.4 Comparisons and Conclusions

Islands have a large syntactic overhead; they require that every variable in a class be labeled with access modes, including the parameters and results of all method signatures. Unique Variables, Flexible Alias Protection, and the Universe Type System have a similar disadvantage, i.e., each variable declaration has to be annotated with an aliasing mode. Furthermore, it seems that a change to the access or alias mode of one parameter or instance variable could have a ripple effect requiring changes to the modes of variables and method signatures in other classes. Leino and Stata's Virgin Objects technique require additional assertions in method specifications so the technique can be enforced modularly.

Balloon types, on the other hand, have considerably less syntactic overhead since they are declared with a single keyword in the definition of a class. However, this low syntactic overhead means that programs must undergo a complex, non-modular static analysis and abstract interpretation to ensure that the Balloon invariant holds. In contrast, our alias control technique can be enforced modularly even though it does not require any additional annotations beyond the standard JML data and method specifications; as mentioned earlier, this was one of the primary goals of our technique.

Islands, Balloons, and Unique Variables are limited because they do not allow abstractions that share objects, a common object-oriented idiom. For example, an abstract data type that shares objects would not be allowed by these techniques, such as an iterator or set of objects. Flexible Alias Protection encapsulates representation objects while allowing argument objects to be statically aliased; however, mutable fields cannot be accessed except by methods of the owner object. The Universe Type System allows sharing of objects, but needs additional restrictions to ensure that invariants of read-only objects hold prior to method calls (Section 7.3). Our technique has read-only references and sharing, but we had to restrict actual parameters in method calls to achieve soundness; loosening these restrictions is an important area of future work.

Islands encapsulate the state of the objects within them. That is, the state of the island cannot change except through calls to methods of the bridge since objects exported via method calls are read-only. Similarly, Flexible Aliasing and Universes encapsulate the state of representation objects. Our technique (like Universes) encapsulates the state of pivot (representation) objects by encapsulating the side-effects applied to these objects while allowing them to be aliased through read-only references. In our technique, all variables and fields that are non-owners are automatically read-only.

Balloons allow external methods to modify internal Balloon objects. Thus modifications by objects external to a Balloon object could cause problems if the changes are unexpected. Therefore, a Balloon class does not necessarily prevent unexpected interference between unrelated classes. Also, if the abstraction might contain non-balloon objects, the members cannot be exported without either deleting the reference or making a deep copy of the exported object.

Balloons, Islands, and Unique Variables (except Boyland's technique) require special operators such as the destructive read, deep copy assignment, or move-assignment, whereas the other techniques do not. Flexible Alias Protection, the Universe Type System, and encapsulation of pivot objects are enforced by a static, modular type system. These techniques can be checked modularly because of the additional assertions and annotations in method signatures and specifications. Our technique can be checked modularly, but without additional annotations or assertions.

A disadvantage of our technique is that references to pivot objects cannot be transferred from a pivot field in one object to a pivot field in another object. The other techniques have a similar disadvantage (unless combined with other restrictions).

Our technique does not encapsulate references to pivot objects, but rather our goal was to encapsulate side-effects to pivot objects by requiring that all changes be initiated through a pivot field.

Also, our technique allows a pivot object to be modified when passed as an argument, but other techniques do not allow this, i.e., they have to be read-only variables or references.

7.5 Contributions and Future Work

7.5.1 Summary of the Problem and Our Solution

In our study, we created formal public and protected specifications for a base class and implemented it; based on this implementation, a code contract for the base class was derived by hand, simulating what our tool would do. We next gave formal public and protected specifications for several new subclasses and studied the problem of how to correctly implement them without access to the superclass code. We found that reasoning about a correct superclass method can only be problematic when there is unexpected aliasing or a new subclass overrides a method, resulting in downcalls. In some cases, when there was no easy or sound way to prevent a problem, we added restrictions to eliminate the problem, i.e., the rules given in Chapters 2 and 3.

As illustrated in Chapters 1 and 3, sound reasoning about the behavior of method calls requires information that would allow users to reason about and prevent or control aliasing. For example, clients cannot reason soundly about programs using public specifications unless the value of the lower-level abstraction (which is usually hidden from clients) remains synchronized with its higher-level value (which is usually public). Our technique deals with aliasing using JML specifications and a static type system (the T-rules given in Chapter 5).

The rules presented in Chapters 2 and 3 generalize our experiences; the T-rules of Chapter 5 provide a formal system for avoiding downcall and aliasing problems. The rules are conservative, because we are assuming that superclass code is not available, and thus the rules can only use information from specifications. The rules allow a programmer to know which methods have to be overridden and when it is safe to call a superclass method. These formal rules form the basis of our proposed tool, which would give warning messages when the rules are violated by a new subclass.

In Chapter 5 we proved the soundness of our technique. In Chapter 6, we provided guidelines for library providers and customizers that greatly simplify reasoning about how to avoid downcall problems, and we described our proposed tool.

7.5.2 Class Library Documentation

One primary goal of this research was to provide enough information in the public and protected specifications and the subclassing contract so programmers could create correct subclasses without the code of the superclass. The subclassing contract and code contract are important new kinds of documentation. The *code contract* contains information derived from the implementation code; in general, this is what our tool would automatically generate. The *subclassing contract* includes the callable and accessible clauses needed for reasoning about new subclasses; these clauses can be part of the code contract or non-code contract (see Section 4.1).

Our formal specifications for superclasses represent the documentation of a class library or framework. Our reasoning technique corresponds to using the documentation. The ability to prove correctness of the subclass is used as a criteria to judge whether these specifications and the reasoning technique are adequate. Since our technique has been shown to be sound, we believe our study provides sound guidance for providing adequate information in user manuals and informal documentation.

7.5.3 The Three-Part Specification

To allow the safe creation of a subclass without using the source code of its superclasses, our technique uses a three part specification, which is incorporated into JML:

1. a public specification used by clients to create and manipulate objects, and by programmers to reason about overriding subclass methods,
2. a protected specification that provides additional information to programmers who want to specialize or extend a class; it includes protected information such as invariants and also specifications for protected instance variables and methods,
3. an automatically-generated code contract that provides important additional information needed by programmers to safely specialize or extend a class.

7.5.4 Contributions

This subsection and the next are an expansion of part of our OOPSLA paper [RL00].

The code contract is an unusual feature of our technique. It is unusual in that it records information about code, as opposed to purely behavioral information. While this is precisely what makes it able to stand in for the code of a library method, it may seem the we are revealing too much detail about the methods. However, we believe that the code contract contains just enough information to safely create subclasses and avoid downcall problems.

As a substitute for source code, such a specification allows a library or framework provider to keep source code secret. But it also functions as a contract with the usual benefits to both reusers and library providers [LG86, Mey92]. Both parties benefit because the specification, and in particular the code contract, abstracts out code details. For reusers, we believe that reading the specification is much less complex than studying the superclass code. For the library provider, it allows some details in the code to change without breaking the contract with reusers. Both parties also gain a more stable contract, since changes to code details do not necessarily break it.

Our technique and proposed tool could also support evolution of a library or framework. Using specifications for an older version of a class, the tool could detect when a new version might invalidate some existing subclass. For example, the tool could give a warning if the code contract of the old version of a class is broken by the new version. A code contract is *broken* if it has additional calls that do not appear in the old version. A broken code contract could invalidate an existing subclass, based

on our rules. The tool could be used to either prevent breaking the contract, or to inform the users of what classes have changed in an incompatible way. In the latter case, reusers could use the tool and the new specifications to correct their subclasses.

Changing the specifications of existing superclass methods and changing the protected invariant of concrete instance variables would also break the superclass's specification. A superclass method specification can be neither weakened nor strengthened; weakening it means the method may no longer behave as expected by clients, and strengthening it may result in a refining subclass method becoming a non-refining method. These violations would not be detected by our tool; library providers would have to notify reusers manually.

A major contribution of our work is its new rules for using such specifications to reason about which methods must be overridden and when super-calls are safe. While the method refinement rule and concrete data refinement rule are based on existing notions of refinement, all the rules rely on the subclassing contract, whose use in reasoning is new with this work. Although the method refinement and concrete data refinement rules would have to be checked manually or with the aid of a theorem prover, the other rules could be checked automatically by our proposed tool.

This reasoning technique is backed up by an analysis of how downcall problems can occur, which is also a contribution of the work.

Our approach to creating subclasses is general in the sense that it does not impose restrictions (other than the assumptions given in subsection 1.6.6) on the implementation of a framework or class library. It identifies potential problems that should be considered by OO programmers using any language. In addition, we have described the details of adapting the rules to Java, using our specification language JML.

While our technique and reasoning method allows for considerable flexibility in inheritance, reusers need considerable knowledge to apply it. As an alternative, we have also offered guidelines for library implementers and reusers (see Chapter 6). These guidelines place greater demands on the library implementer, but make only limited demands on the customizer.

Our technique for preventing unexpected side-effects is another important contribution of our work. We have shown that pivot objects can be protected from unwanted side-effects with a few simple rules (Section 3.5) that can be statically and modularly enforced (Section 5.1). Furthermore, this can be done without the large syntactic overhead of other techniques or a complicated whole program analysis.

7.5.5 Future Work

There are several areas for future work in addition to those mentioned earlier. One area is to extend our ideas to languages with different kinds of inheritance, such as Beta's [MMPN93]. Another area is to relate concrete data refinement to refactoring [Opd92].

Our proposed tool is also important future work. Recall that this tool would use superclass specifications and the specification of a new subclass to list superclass methods that must be overridden based on the rules. It would also statically generate the code contract of the new subclass, and check for violations of the rules. Building this tool is important future work because it would catch and help prevent potential errors prior to execution.

A case study is also important future work. It is important to find out how practical or limiting our technique is for real world systems. As mentioned earlier, another important direction for future work is the loosening of the restrictions on actual parameters that were necessary for soundness.

A: RULES FROM CHAPTER 2

A.1: Overriding Rules

Additional Side-Effects Overriding Rule. Let *S* be a subclass of *C*. If *S* specifies that method *C*::*M* can have additional side-effects on field *W* or if method *C*::*M* makes a self-call down to a method that may have additional side-effects on *W*, then *C*::*M* must be overridden.

Invariant Overriding Rule. Let *S* be a subclass of *C*. Let *V* be a concrete instance variable visible in *S*. Let *S* specify an invariant that accesses and constrains the value of *V*. If superclass method *C*::*M* can assign to *V*, then *C*::*M* must be overridden.

Callback Cycle Overriding Rule. Let *P* be an overriding method in a subclass. If *P* self-calls a superclass method *M* that self-calls back directly or indirectly to *P*, then *M* must be overridden.

A.2: Invalidation Rules

Additional Side-Effects Invalidation rule. Let *S* be a subclass of *C*. Let *S* specify that superclass method (or constructor) *C*::*M* can have additional side-effects on field *W*. If *C*::*M* self-calls down to a method *S*::*N* that is allowed to modify *W*, then *C*::*M* may not be super-called by any method (or constructor) of *S*.

Invariant Invalidation Rule. Let *S* be a subclass of *C*. Let *V* be a concrete instance variable visible in *C*. Let *S* specify an invariant that accesses and constrains the value of *V*. If superclass method *C*::*M* can assign to *V* and it makes a self-call, then *C*::*M* may not be super-called by any method (or constructor) of *S*.

Callback Cycle Invalidation Rule. Let *S* be a subclass of *C*. Let *P* be an overriding method in *S*. If a superclass method *M* directly or indirectly self-calls down to method *S*::*P*, then *C*::*M* cannot be super-called by *S*::*P*.

Constructor Invariant Invalidation Rule. Let *S* be a subclass of *C*. If *S* specifies a subclass invariant and constructor *C*::*M* self-calls down to a method *S*::*N*, then *C*::*M* may not be super-called by constructors of *S*.

Constructor Initialization Invalidation Rule. Let *S* be a subclass of *C*. Let *W* be an instance variable directly or indirectly declared in *S*. If constructor *C*::*M* self-calls down to a method *S*::*N* that directly or indirectly accesses the value of *W*, then *C*::*M* may not be super-called by constructors of *S*.

A.3: Special Rules

Mandatory Super-Call Rule. Let *S* be a subclass of *C*. Let *V* be an instance variable declared in *C*. Let *V*'s data group contain private variables. If *S* has to implement a new or overriding method (or constructor) *M* that modifies *V*, then *S*::*M* can only modify *V* (along with the private variables in *V*'s data group) by directly or indirectly super-calling methods (or constructors) of *C*.

Super-Call Authorization Rule. Let *S* be a subclass of *C*. A superclass method *C*::*M* may only be called by subclass method *S*::*P*, if *C*::*M* has not been invalidated for *S*::*P*.

A.4: Other Rules (not applicable to JML)

Data Refinement Overriding Rule. Let *S* be a subclass of *C*. A method, *M*, must be overridden if (i) *C*::*M* makes a direct self-access or object-access to a concrete variable *V* that is data refined by *S* and (ii) the part of *C*'s invariant[1] that concerns *V* is not maintained by *S*.

Data Refinement Invalidation Rule. A superclass method must not be super-called if it had to be overridden based on the data refinement overriding rule.

Non-Refining Method Overriding Rule. A superclass method must be overridden if it makes a direct self-call or a subclass object-call to a method that has been overridden by a non-refining method.

Non-Refining Method Invalidation Rule. A superclass method must not be super-called if it makes a direct self-call or a subclass object-call to a method that has been overridden by a non-refining method.

Unoverrideable Method Rule. Let *M* be a non-public superclass method that is not (object-) called by methods of an unrelated class. If *M* cannot be overridden and is invalidated by the new subclass, then all methods that directly call *M* must be overridden and *M* cannot be super-called by subclass methods.

Method Refinement Overriding Rule. Let *S* be a subclass of *C*. If the specification of *S*::*M* refines the behavior of superclass method *C*::*M*, then *C*::*M* must be overridden.

---

1. The represents clause specifies an invariant relationship between a model field and concrete fields; therefore, we consider the represents clause to be a part of the superclass invariant that needs to be maintained unless the subclass is doing a concrete data refinement.

B: RULES FROM CHAPTER 3

B.1: Declaring Pivot Fields

Pivot Declaration Rule. Let *x.V* be an instance field indirectly declared in type *C* or one of *C*'s supertypes. If *x* is a concrete field and *x.V* is accessed on the right-hand side of a represents clause or in a predicate clause in the specification of *C*, then *x.V* must be a member of (i.e. mapped into) a data group visible to *C*.

Predicate Clause Access Rule. Let class *S* be *C* or a subtype of *C*. Let *F* be a model field with a reference type that is directly declared in *C*. If *F*'s represents clause accesses a non-pivot object *V*, then fields of model object *F* cannot be accessed by predicate clauses in *C* and *S*.

Predicate Clause Access Rule???. Let class *S* be *C* or a subtype of *C*. Let *F* be a model field with a reference type that is directly declared in *C*. If predicate clauses in *C* and *S* access a field *F.x*, then *F.x* cannot access fields of a non-pivot object [[[may not be necessary]]].

Model Field Access Rule. Let class *S* be *C* or a subtype of *C*. Let *F* be a model field with a reference type that is directly declared in *C*. If *F*'s represents clause maps a concrete field *V* to *F*, i.e., if they both denote the same object, then fields of model object *F* can be accessed by predicate clauses in *C* and *S*, otherwise fields of *F* cannot be accessed by predicate clauses.

Represents Clause Access Rule. Expressions occurring on the right side of a represents clause must follow the syntax given in Figure 3.10.

Assignable Clause Rule. A field of a model object cannot be directly accessed in an assignable or maps clause.

B.2: Specifying and Controlling Side-Effects

Model field Data Group Rule. Let *V* be a concrete instance field directly or indirectly declared in *C* or in a superclass of *C*. Let *F* be an instance model field directly declared in *C* or in a supertype of *C*. If *V* is accessed on the right-hand side of *F*'s represents clause, then either (i) *V* must be a member of data group *F* or (ii) *V* must be a member of at least one data group and *F* must be a member of all such data groups containing *V*.

Assignable Data Group Rule. Let *V* be a concrete instance field directly or indirectly declared in *C* or in a superclass of *C*. Let *F* be an instance model field directly declared in *C* or in a supertype of *C*. If *V* is a member of data group *F* and *V* is assignable in an instance method *M*, then *F* must also be assignable in *M*.

B.3: Protecting Pivot Objects

Owner Variable Rule. When a method is allowed to assign to fields of the receiver or formal parameter object, then the corresponding actual parameter must be an owner variable, `null`, or a new object constructor call.

Pivot Assignment Rule. When a pivot field or formal parameter is the left operand (target) of an assignment statement, then the right operand must be `null` or a new object constructor call, unless the type of the target variable is immutable or a primitive type.

Actual Parameter Aliasing Rule. An argument object of a method call cannot have assignable fields if the called method directly or indirectly accesses those assignable fields through a different access path.

C: RULES FROM CHAPTER 5

Actual Parameter Aliasing Rule′. Two argument objects to a method call cannot have aliased fields if those fields are assignable in the called method.

Subclass Invariant Rule: Let *S* be a subclass of *C*. Let *V* be a concrete instance variable visible in *C*. If superclass method *C*::*M* is unoverrideable and can assign to *V*, then *S* cannot specify an invariant that accesses and constrains the value of *V*.

Additional Side-Effects Rule: Let *S* be a subclass of *C*. Let *V* be a concrete instance variable declared in *S*. If superclass method *C*::*M* is unoverrideable, then *S* cannot allow *C*::*M* to assign to *V*.

BIBLIOGRAPHY

[AdB90]America, P. and de Boer, F. A sound and complete proof theory for SPOOL. Technical Report 505, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., May 1990.

[AKC02]Aldrich, J., Kostadinov, V., and Chambers, C. Alias annotations for program understanding. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, volume 37(11) of *ACM SIGPLAN Notices*, pages 311–330. ACM, November 2002.

[Alm97]Almeida, P. S. Balloon types: Controlling sharing of state in data types. In Akcsit, M. and Matsuoka, S., editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, New York, NY, June 1997.

[Ame91]America, P. Designing an object-oriented programming language with behavioural subtyping. In de Bakker, J. W., de Roever, W. P., and Rozenberg, G., editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.

[Bak95]Baker, H. G. 'use-once' variables and linear objects – storage management, reflection, and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[BCC+03]Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G. T., Leino, K. R. M., and Poll, E. An overview of JML tools and applications. In Arts, T. and Fokkink, W., editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[BCC+95]Bruce, K., Cardelli, L., Castagna, G., Group, T. H.O., Leavens, G. T., and Pierce, B. On binary methods. Technical Report 95-08a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1995. Appears in *Theory and Practice of Object Systems*. Volume 1, Number 3. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[BDF+04]Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[BN05]Banerjee, A. and Naumann, D. A. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, November 2005.

[BNR01]Boyland, J., Noble, J., and Retert, W. Capabilities for sharing. In Knudsen, J. L., editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 1–27, Berlin, June 2001. Springer-Verlag.

[Boy01]Boyland, J. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[BvW98]Back, R.-J. and von Wright, J. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[CC00]Chen, Y. and Cheng, B. H. C. A semantic foundation for specification matching. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, New York, NY, 2000.

[CD02]Clarke, D. and Drossopoulou, S. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, volume 37(11) of *ACM SIGPLAN Notices*, pages 292–310. ACM, November 2002.

[CLCM00]Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, NY, October 2000. ACM.

[Cli01]Clifton, C. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. The author's masters thesis.

[CNP01]Clarke, D. G., Noble, J., and Potter, J. M. Simple ownership types for object containment. In Knudsen, J. L., editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 53–76, Berlin, June 2001. Springer-Verlag.

[CPN98]Clarke, D. G., Potter, J. M., and Noble, J. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.

[CW03]Clarke, D. and Wrigstad, T. External uniqueness is unique enough. In Cardelli, L., editor, *ECOOP 2003 — Object-Oriented Programming: 17th European Conference, Darmstadt, Germany*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200, Berlin, July 2003. Springer-Verlag.

[DL96]Dhara, K. K. and Leavens, G. T. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[DLN98]Detlefs, D. L., Leino, K. R. M., and Nelson, G. Wrestling with rep exposure. SRC Research Report 156, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, July 1998.

[DM05]Dietl, W. and Müller, P. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[Edw96]Edwards, S. H. Representation inheritance: A safe form of "white box" code inheritance. In *Fourth International Conference on Software Reuse*, pages 195–204. IEEE Computer Society Press, April 1996.

[GB99]Greenhouse, A. and Boyland, J. An object-oriented effects system. In Guerraoui, R., editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 205–229. Springer-Verlag, New York, NY, June 1999.

[GHG+93]Guttag, J. V., Horning, J. J., Garland, S., Jones, K., Modet, A., and Wing, J. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[GM94a]Gardier, P. H. B. and Morgan, C. A single complete rule for data refinement. In Morgan and Vickers [MV94], pages 111–126.

[GM94b]Gardiner, P. H. B. and Morgan, C. Data refinement of predicate transformers. In Morgan and Vickers [MV94], pages 71–84.

[Heh93]Hehner, E. C. R. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hes92]Hesselink, W. H. *Programs, Recursion, and Unbounded Choice*, volume 27 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, NY, 1992.

[HK00]Huizing, K. and Kuiper, R. Verification of object-oriented programs using class invariants. In Maibaum, E., editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, 2000.

[HLR+99]Hakonen, H., Leppänen, V., Raita, T., Salakoski, T., and Teuhola, J. Improving object integrity and preventing side effects via deeply immutable references. In *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, FUSST'99*, pages 139–150, 1999.

[HLW+92]Hogg, J., Lea, D., Wills, A., deChampeaux, D., and Holt, R. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, April 1992.

[Hoa69]Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa72]Hoare, C. A. R. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Hog91]Hogg, J. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[KdRB91]Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Mass., 1991.

[KL92]Kiczales, G. and Lamping, J. Issues in the design and documentation of class libraries. *ACM SIGPLAN Notices*, 27(10):435–451, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[KT01]Kniesel, G. and Theisen, D. Jac — acess right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, 2001.

[Lam93]Lamping, J. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[LBR98]Leavens, G. T., Baker, A. L., and Ruby, C. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. `http://www-dse.doc.ic.ac.uk/char'176sue/oopsla/cfp.html`.

[LBR01]Leavens, G. T., Baker, A. L., and Ruby, C. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, December 2001. This is an obsolete version.

[Lea06]Leavens, G. T. JML's rich, inherited specifications for behavioral subtypes. In Liu, Z. and Jifeng, H., editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[Lei95]Leino, K. R. M. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Lei98]Leino, K. R. M. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[LG86]Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[LG88]Lucassen, J. M. and Gifford, D. K. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 47–57. ACM, January 1988.

[LH89]Lieberherr, K. J. and Holland, I. M. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989.

[LM04]Leino, K. R. M. and Müller, P. Object invariants in dynamic contexts. In Odersky, M., editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, June 2004.

[LM05]Leino, K. R. M. and Müller, P. Modular verification of static class invariants. In Fitzgerald, J., Hayes, I. J., and Tarlecki, A., editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, July 2005.

[LM06]Leino, K. R. M. and Müller, P. A verification methodology for model fields. In *European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[LN00]Leino, K. R. M. and Nelson, G. Data abstraction and information hiding. Technical Report 160, Compaq Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, 2000. To appear in TOPLAS.

[LN02]Leino, K. R. M. and Nelson, G. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[LP06]Lu, Y. and Potter, J. Protecting representation with effect encapsulation. In *ACM Symposium on Principles of Programming Languages*, pages 359-371, 2006.

[LPHZ02]Leino, K. R. M., Poetzsch-Heffter, A., and Zhou, Y. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 246–257, New York, June 17–19 2002. ACM Press.

[LR91]Landi, W. and Ryder, B. G. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando*, pages 93–103. ACM, January 1991.

[LS99]Leino, K. R. M. and Stata, R. Virginity: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, Apr 1999.

[Luc97]Lucas, C. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 1997.

[LW93]Liskov, B. and Wing, J. M. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[LW94]Liskov, B. and Wing, J. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95]Leavens, G. T. and Weihl, W. E. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mey88]Meyer, B. *Object-oriented Software Construction*. Prentice Hall, New York, NY, 1988.

[Mey92]Meyer, B. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Mey97]Meyer, B. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[Mez97]Mezini, M. Maintaining the consistency and behavior of class libraries during their evolution. *ACM SIGPLAN Notices*, 32(10):1–21, October 1997. *Conference Proceedings of OOPSLA '97*.

[MG90]Morgan, C. and Gardiner, P. H. B. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, May 1990.

[Min96]Minsky, N. H. Towards alias-free pointers. In Cointe, P., editor, *ECOOP '96 – Object-Oriented Programming: 10th European Conference, Linz Austria*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, July 1996. Springer-Verlag.

[MMPN93]Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA programming Language*. Addison-Wesley Inc, 1993.

[Mor94]Morgan, C. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[MPH00]Müller, P. and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.

[MPH01]Müller, P. and Poetzsch-Heffter, A. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001. Available from rlwww.informatik.fernuni-hagen.de/pi5/publications.html.

[MPHL05]Müller, P., Poetzsch-Heffter, A., and Leavens, G. T. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, March 2005.

[MPHL06]Müller, P., Poetzsch-Heffter, A., and Leavens, G. T. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, October 2006.

[MS98]Mikhajlov, L. and Sekerinski, E. A study of the fragile base class problem. In Jul, E., editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, July 1998.

[Mül01]Müller, P. *Modular Specification and Verification of Object-Oriented programs*. PhD thesis, FernUniversität Hagen, Germany, March 2001.

[Mül02]Müller, P. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. The author's Ph.D. Thesis.

[MV94]Morgan, C. and Vickers, T., editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, NY, 1994.

[NVP98]Noble, J., Vitek, J., and Potter, J. Flexible alias protection. In Jul, E., editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[Opd92]Opdyke, W. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[PH97]Poetzsch-Heffter, A. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[PHM99]Poetzsch-Heffter, A. and Müller, P. A programming logic for sequential Java. In Swierstra, S. D., editor, *European Symposium on Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[PK90]Perry, D. E. and Kaiser, G. E. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, Jan/Feb 1990.

[RL00]Ruby, C. and Leavens, G. T. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, October 2000.

[RL03]Raghavan, A. D. and Leavens, G. T. Desugaring JML method specifications. Technical Report 00-03d, Iowa State University, Department of Computer Science, July 2003. This is an obsolete version.

[SG95]Stata, R. and Guttag, J. V. Modular reasoning in the presence of subclassing. *ACM SIGPLAN Notices*, 30(10):200–214, October 1995. *Proceedings of OOPSLA '95 Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

[SGM02]Szyperski, C., Gruntz, D., and Murer, S. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, second edition edition, 2002.

[Sko02]Skoglund, M. Sharing objects by read-only references. In Kirchner, H. and Ringeissen, C., editors, *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 457–472, New York, NY, 2002. Springer-Verlag.

[SLMD96]Steyaert, P., Lucas, C., Mens, K., and D'Hondt, T. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 268–285. ACM Press, October 1996. ACM SIGPLAN Notices, Volume 31, Number 10.

[Sta97]Stata, R. Modularity in the presence of subclassing. Technical Report 145, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, Apr 1997. Order from src-report@pa.dec.com or ftp from gatekeeper.dec.com.

[Szy98]Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[vO01]von Oheimb, D. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[VB01]Vitek, J. and Bokowski, B. Confined types in Java. *Software — Practice and Experience*, 31(6):507–532, 2001.

[Wil91]Wills, A. Capsules and types in Fresco: Program validation in Smalltalk. In America, P., editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, New York, NY, 1991.