Safely Creating Correct Subclasses without Seeing Superclass Code

Clyde Ruby and Gary T. Leavens

TR #00-05d
April 2000, revised April, June, July 2000

Keywords: Downcalls, subclass, semantic fragile subclassing problem, subclassing contract, specification inheritance, method refinement, Java language, JML language.
1999 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, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation, Restructuring, reverse engineering, and reengineering; 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;

To appear in OOPSLA 2000, Minneapolis, Minnesota, October 2000.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Safely Creating Correct Subclasses without Seeing Superclass Code

Clyde Ruby and Gary T. Leavens*
Department of Computer Science
Iowa State University
226 Atanasoff Hall, Ames, IA 50011 USA
+1 515 294 1580
{ruby,leavens}@cs.iastate.edu

ABSTRACT

A major problem for object-oriented frameworks and class libraries is how to provide enough information about a superclass, so programmers can safely create new subclasses without giving away the superclass's code. Code inherited from the superclass can call down to methods of the subclass, which may cause nontermination or unexpected behavior. We describe a reasoning technique that allows programmers, who have no access to the code of the superclass, to determine both how to safely override the superclass's methods and when it is safe to call them. The technique consists of a set of rules and some new forms of specification. Part of the specification would be generated automatically by a tool, a prototype of which is planned for the formal specification language JML. We give an example to show the kinds of problems caused by method overrides and how our technique can be used to avoid them. We also argue why the technique is sound and give guidelines for library providers and programmers that greatly simplify reasoning about how to avoid problems caused by method overrides.

Keywords

Downcalls, subclass, semantic fragile subclassing problem, subclassing contract, specification inheritance, method refinement, Java language, JML language.

1. INTRODUCTION

Our long term goal is to discover what makes good quality documentation of frameworks and class libraries. Usually, not enough documentation is provided for programmers to write a subclass without studying the source code of its superclasses. One benefit of sufficient and unambiguous documentation of object-oriented (OO) class libraries and frameworks 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 problem we address in this paper is what information is needed in such documentation, and how to use it to create subclasses that do not exhibit problems such as nontermination or unexpected behavior.

A correct superclass method can only be problematic when a new subclass overrides some methods, leading to downcalls. A *downcall* occurs when a superclass method calls a method that is overridden in the subclass. We also say that the superclass "calls down to" the overridden method, because we visualize the class diagram with subclasses below superclasses. Overriding is important, because no problems occur when a superclass calls one of its own methods. On the other hand, when the superclass calls down to an overridden method, this overriding subclass method may behave differently than the superclass method expects.

Downcalls are also related to callbacks [39, p. 107]. Callbacks can happen when a subclass method calls a superclass method that makes a downcall; the downcall then is also a callback, because it leads back to the subclass. However, not every callback involves a superclass calling down to an overridden method.

Downcall problems are related to the so-called semantic fragile base class problem [27] [39, pp. 102-104], which is concerned with how to change superclasses without invalidating existing subclasses. The problem we are solving might be called the *semantic fragile subclassing problem*, which deals with how to create a valid subclass; that is, how to override superclass methods in such a way that the subclass is free from downcall problems.

To understand what information is needed by programmers to avoid downcall problems, we study 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

*The work of both Ruby and Leavens was supported in part by the US National Science Foundation under grant CCR-9803843.

To appear in *OOPSLA 2000* Minneapolis, Minnesota, October 2000.

used as a criteria to judge whether these specifications and the reasoning technique are adequate. Therefore, we believe our study provides sound guidance for providing adequate information in user manuals and informal documentation. Some discussion of this is found in Section 7.4.

For our purposes, the specifications given for the classes in a library or framework have three parts:

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

The subclassing contract would be automatically generated by a planned tool. Section 3 gives more details about the contents of these specifications.

In Section 4 we give examples of downcall problems and explain our rules for reasoning about how to avoid each particular problem. In a few cases, when there is no easy or sound way to prevent the problem, we give restrictions on superclass code that would eliminate the problem. In our study, we created formal public and protected specifications for a base class and implemented it; based on this implementation, a subclassing 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.

The rules presented in Section 4 generalize our experience and provide a formal system for avoiding downcall 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 and when it is safe to call a superclass method. They form the basis of our proposed tool, which would give warning messages when the rules are violated by the subclass. In Section 5 we argue the soundness of these rules. In Section 6 we describe our proposed tool. In Section 7 we give guidelines for library providers and programmers that can greatly simplify reasoning about how to avoid downcall problems. In Section 8 we discuss related work, and we offer some conclusions in Section 9.

2. ASSUMPTIONS

Our investigation focuses primarily on the issues involved in code inheritance and subclassing in a single-dispatch[1] OO programming language. We do not consider other language-specific features such as nested classes or exceptions.

[1]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.

We make several simplifying assumptions that are important for the soundness of our approach. We assume that the superclass's code does not name the new subclass, since a library class would not know about new subclasses. We assume that methods do not access instance variables[2], even public ones, in objects of unrelated classes. 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. We assume that methods do not temporarily change and then restore instance variables around calls to public methods, as in Figure 8 below. Finally, we assume that classes do not have mutually recursive methods from unrelated classes.

3. JML SPECIFICATIONS

Although our ideas and techniques are independent of any particular programming language, for concreteness in examples we use Java and the Java Modeling Language (JML) [14, 15] as our programming and specification notations. JML blends the Eiffel [24] and Larch [8] traditions, and like Eiffel uses Java expressions within assertions.

3.1 Public Specification

Figure 1 gives a formal public specification of a simple `Point` class in JML. 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.

The modifier `model` in a declaration means that the declaration is for use in specifications and need not be part of the implementation. For example, `xCoord` is an instance variable used only for specifying method behavior. Similarly, the `model` modifier means that `distance` is a specification-only method. (Note that it is enclosed entirely in a JML annotation.) A method may only be used in assertions if it does not have side-effects; the modifier `pure` specifies that a method, like `distance`, has no side-effects, and thus can be used in assertions.

In JML method specifications, preconditions start with the keyword `requires`, postconditions with `ensures`, and frame conditions with `modifiable`. A `modifiable` clause specifies the variables that may be modified by the method; for example, in the method `moveX` of `Point`, only the model variables `xCoord` and `oldX`, and the variables on which they depend (see below) may be modified.

The "`normal`" in the `public_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`" prefix 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.

3.2 Protected Specification

[2]We use the term *instance variable* for what Java calls a non-static data field.

```
package edu.iastate.cs.jml.paper;

public class Point
{
   //@ public model int xCoord;      // model variables
   //@ public model int yCoord;
   //@ public model int oldX;
   //@ public model int oldY;

  /*@  public_normal_behavior
    @    ensures: \result == Math.abs(xDist) + Math.abs(yDist);
    @ public pure model int distance(int xDist, int yDist);   @*/

  /*@  public_normal_behavior
    @   modifiable: xCoord, yCoord, oldX, oldY;
    @   ensures: xCoord == initX && yCoord == initY
    @             && oldX == initX && oldY == initY;   @*/
  public Point(int initX, int initY);

  /*@  public_normal_behavior
    @   ensures: \result == xCoord;   @*/
  public /*@ pure @*/ int getX();

  /*@  public_normal_behavior
    @    ensures: \result == yCoord;   @*/
   public /*@ pure @*/ int getY();

  /*@  public_normal_behavior
    @   modifiable: xCoord, yCoord, oldX, oldY;
    @   ensures: xCoord == newX && yCoord == newY
    @             && oldX == \old(xCoord) && oldY == \old(yCoord);   @*/
   public void move(int newX, int newY);

  /*@  public_normal_behavior
    @   modifiable: xCoord, oldX;
    @   ensures: xCoord == newX && oldX == \old(xCoord);   @*/
   public void moveX(int newX);

  /*@  public_normal_behavior
    @   modifiable: yCoord, oldY;
    @   ensures: yCoord == newY && oldY == \old(yCoord);   @*/
   public void moveY(int newY);

  /*@  public_normal_behavior
    @   ensures: \result == distance(xCoord - oldX, yCoord - oldY);   @*/
   public /*@ pure @*/ int distanceMoved();

  /*@  public_normal_behavior
    @    requires: p != null;
    @    ensures: \result == distance(p.xCoord - xCoord, p.yCoord - yCoord);   @*/
   public /*@ pure @*/ int distanceTo(Point p);
}
```

Figure 1: Public specification of `Point` in file `Point.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.

```
package edu.iastate.cs.jml.paper;

//@ refine: Point <- "Point.jml-refined";

public class Point
{
   protected int x, y;
   protected int deltaX, deltaY;

   //@ protected depends: xCoord -> x;
   //@ protected represents: xCoord <- x;
   //@ protected depends: yCoord -> y;
   //@ protected represents: yCoord <- y;

   //@ protected depends: oldX -> deltaX;
   //@ protected represents: oldX <- (x - deltaX);
   //@ protected depends: oldY-> deltaY;
   //@ protected represents: oldY <- (y - deltaY);

   /*@  protected_normal_behavior
     @   requires: p != null;
     @   modifiable: xCoord, yCoord, oldX, oldY;
     @   ensures: this.xCoord == p.xCoord
     @         && this.yCoord == p.yCoord
     @         && oldX == p.xCoord
     @         && oldY == p.yCoord;
     @*/
   protected Point(Point p);
}
```

Figure 2: Protected specification of class `Point` in file Point.jml.

The protected specification provides additional documentation that is needed by programmers when specializing or extending a class. It specifies the parts of the class visible to subclasses, including protected instance variables and the behavior of protected constructors and methods. It will include 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 2 gives the protected specification of class `Point`; it specifies the relation between the public and protected instance variables and gives the specification of a protected constructor. The protected specifications in Figure 2 are added to the public specifications in Figure 1 because of the `refine` clause found after the package declaration in Figure 2.

The `depends` and `represents` clauses specify the relationship between variables. Typically the variables are defined in two different places such as the superclass and subclass, or, as in this case, in the public and protected specifications. In this example, they specify the relation between the public specification-only variables and the protected concrete variables of the class. The term *concrete* means part of the implementation of a class. The public variables are needed 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.

The `depends` clause specifies a dependency relationship that controls which variables can be modified by methods [17, 18]. For example, in Figure 2, the first `depends` clause says that `xCoord` depends on the protected instance variable `x`; this allows `x` to be modified by a method whenever `xCoord` is modifiable. Without this `depends` clause `x` could not be modified by any of the methods of class `Point`. Similarly, the other `depends` clauses control which concrete variables can be modified by method implementations.[3]

The `represents` clause also specifies a relationship between variables; it specifies how a variable can be derived from one or more other variables. For example, the third `represents` clause specifies how the value of `oldX` can be derived from the concrete variables `x` and `deltaX`.

The keywords that introduce specifications restrict the scope of variables. E.g., `protected_normal_behavior` means that both public and protected instance variables and methods are in scope in the specification that follows it; this is because of prefix "`protected`." Nevertheless, the specification of the protected constructor in Figure 2 uses public model variables even though protected variables are in scope; this makes its specification independent of the implementation.

3.3 Subclassing Contract

The purpose of the subclassing contract is to give programmers additional information they can use to avoid downcall problems. The subclassing contract must be generated for all methods and constructors involved in reasoning about downcalls; thus they are generated for some methods and constructors that may not seem obviously needed at first, such as non-public methods and constructors. Figure 3 shows the subclassing contract of class `Point`. The subclassing contract includes the `callable` and `accessible` clauses that would be automatically derived from implementation code by our proposed tool.

3.3.1 The Callable Clause

The `callable` clause lists the signatures of methods (and constructors) directly called by the method being specified. Because constructor calls can also cause downcall problems, we include constructors when we use the term "method." A method M *directly calls* another method N if the code for M has an expression that calls N, such as "`N()`." If N also directly calls method P, then M *indirectly calls* P.

Various kinds of calls are 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. In the presence of static overloading, as in Java, it is also necessary that the `callable` clause distin-

[3]Note that the `depends` clause does not control what names are in scope in a specification.

```
package edu.iastate.cs.jml.paper;
//@ refine: Point <- "Point.jml";
public class Point {
  /*@ also
    @ subclassing_contract
    @   accessible: x, y;
    @   callable: \nothing;
    @*/
   public Point(int initX, int initY);

  /*@ also
    @ subclassing_contract
    @   accessible: x, y;
    @   callable: moveX(int), moveY(int);
    @*/
   public void move(int newX, int newY);

  /*@ also
    @ subclassing_contract
    @   accessible: x;
    @   callable: \nothing;
    @*/
   public void moveX(int newX);

  /*@ also
    @ subclassing_contract
    @   accessible: y;
    @   callable: \nothing;
    @*/
   public void moveY(int newY);

  /*@ also
    @ subclassing_contract
    @   accessible: p.x, p.y,
    @               x, y, deltaX, deltaY;
    @   callable: distanceMoved(),
    @             move(int, int);
    @*/
   public int distanceTo(Point p);

  /*@ also
    @ subclassing_contract
    @   accessible: x, y, deltaX, deltaY;
    @   callable: \nothing;
    @*/
   public int distanceMoved();

  /*@ also
    @ subclassing_contract
    @   accessible: p.x, p.y;
    @   callable: \nothing;
    @*/
   protected Point(Point p);

   // ... other methods omitted
}
```

Figure 3: Subclassing contract for class `Point` from `Point.refines-jml`.

guishes which method (or constructor) is being called. Thus argument types are included along with the method name.

Downcall and callback problems can occur when an instance method M passes its implicit receiver parameter, `this`, as an argument to some method N (which could be the same as M). Some calls pass `this` implicitly, and some pass it explicitly. A call in which `this` (the current method's receiver), is explicitly passed as an argument will be referred to as a *this-argument call*. By "explicitly" we do not mean that a this-argument call must literally include "`this`" as an actual argument; calls in which `this` is passed inside a data structure, such as an array, are also considered to be this-argument calls.

We distinguish two kinds of calls in which `this` is passed implicitly: super-calls and self-calls.

A *super-call* is a call where either the receiver is the built-in variable `super`, such as "`super.move(u,v)`" in Java, or a *superclass constructor call*, such as "`super()`" in Java. In a `callable` clause, a super-call to a method will be recorded with a signature like "`super.move(int,int)`," and a superclass constructor will be recorded as "`super()`." (In languages with multiple-inheritance, like C++, one could also refer to a specific superclass in a super-call, for example, "`Point::move(u,v)`," which would appear as "`Point::move(int,int)`.") Superclass method calls and superclass constructor calls are both included in our 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.

A *self-call* is a call such as "`move(u,v)`" which is sugar for "`this.move(u,v)`," i.e., a call in which `this` is the receiver object. In a `callable` clause, a self-call will appear as "`move(int,int)`."

Recall that a this-argument call is one that receives `this` as an argument. All this-argument calls can indirectly access the calling method's receiver and use it to make downcalls. Therefore, three other kinds of calls must be distinguished in the `callable` clause: object-calls, static-calls, and new object constructor calls. These three kinds of call can only access the calling method's `this` if it is passed as an argument to them; thus they can only be involved in downcall problems via this-argument calls.

An *object-call* is a call in which the receiver is an object other than `super` or `this`; e.g., "`p.move(u,v)`" is an object-call. Such an object-call is recorded with a signature like "`Point.move(int,int)`," where `Point` is the static type of the receiver.

A *static-call* is a call to a static method; for example, the call "`Math.abs(x)`" is a static-call in Java. In a `callable` clause, the signature of a static-call has the same form as an object-call. However, this notation is not ambiguous because we assume that static and non-static method names cannot conflict in the same class.

A *new object constructor call* is an expression, for example "`new Point(i,j)`" in Java or C++, that creates a new object by invoking a constructor. In a `callable` clause, a new object constructor call has a signature such as "`new Point(int,int)`."

The above definitions assume perfect knowledge of aliasing; that is, one may only be able to decide at run-time what kind of call is being made by a specific piece of program text. However, when our proposed tool is statically generating the `callable` clause, it will have only approximate knowledge of aliasing. To compensate, a call like "`p.move(u,v)`" will appear in a `callable` clause twice if `p` is a possible alias of the implicit parameter `this`; that is, it will appear once as a self-call and once as an object-call.

3.3.2 The Accessible Clause

The `accessible` clause of a method M in a class C provides the list of instance variables of objects of type C (including inherited and non-public variables) that are directly accessed by M. In addition, fields of other objects whose type could be C (or a subclass)[4] are included in the `accessible` clause. Accesses to instance variables of the receiver (`this`) will be called *self-accesses*, and accesses to fields of other potential subclass objects will be called *object-accesses*. Self-accesses and object-accesses will be distinguished in the `accessible` clause. As for the `callable` clause, if the tool does not have exact aliasing information, accesses may be listed twice.

For example, the subclassing contract of method `distanceTo` in Figure 3 would be generated from the implementation shown in Figure 8 below. Note that local variables accessed in that method are not included in the `accessible` clause.

Inheritance of subclassing contracts is different from inheritance of the rest of the specification, because it is derived from and reflects the code of each method or constructor. So when a method is inherited without change from its superclass (by not overriding the method), then the subclass inherits the subclassing contract from the superclass for that method. However, when a method is overridden by the subclass, that method's subclassing contract is just derived from the subclass's code.

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 comes with a tool that can automatically combine specifications from various files into such a complete specification [34].)

4. CREATING NEW SUBCLASSES

This section gives a set of rules for dealing with potential problems that should be considered by programmers when creating subclasses. The rules only use information in the subclassing contract and the public and protected specifications.

Two kinds of rules are given: rules for determining which methods must be overridden and rules for determining when super-calls are safe.

The intent of the *overriding rules* is to determine which methods to override, again assuming the superclass's code is not available. The overriding rules are discussed in Subsections 4.1–4.6. 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 specified, other than those caused by super-calls, are avoided when the overriding rules are followed and our assumptions have also been followed.

The intent of the *super-call authorization rule*, discussed in Subsection 4.7, is to determine when it is safe to make a super-call. Its application is closely related to the overriding rules, but its purpose is to prevent callback problems involving super-calls.

The subsections describing these rules are followed by a discussion of some consequences. This discussion appears in Subsection 4.8.

Subclass methods can call overridden methods via super-calls and new object constructor calls. Therefore, when necessary to avoid possible ambiguities, methods and constructors will 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++.

4.1 New Instance Variables

This subsection describes an overriding rule that prevents problems resulting from side-effects to new instance variables. It introduces the concepts of codependency and additional side-effects.

For example, consider Figure 4, which contains the public specification of `PointPlusTotal`, a new subclass of `Point`. This subclass adds method `getTotalDistance` and adds to the specifications for `move`, `moveX`, and `moveY`. The protected specification is given in Figure 5. This new subclass adds a new concrete instance variable named `_totalDist_` that tracks the total distance a point object has been moved since it was first created.

The `depends` clauses in Figure 4 allow the model variable `totalDist` to change when the model variables in `Point` change. The `depends` clause in Figure 5 also transitively allows the concrete variable `_totalDist_` to change when the public model variables change in `Point`. This allows the methods that move the point to modify the new concrete variable. The concrete variable is modified as specified in the `ensures` clause of these methods, taking into account the `represents` clause in Figure 5. (This `represents` clause says that the value of totalDist is given by the value of `_totalDist_`.) Thus all methods that move the point

[4]In a single-inheritance language, like Java, the only types that are potentially subtypes of C are C itself and any known subclasses. In a language with multiple inheritance, like C++, one would have to be more conservative, since an unrelated class could, through later use of multiple inheritance, have a subclass that is also a subclass of C.

will have to be overridden. The rule below ensures that the necessary methods have been overridden.

However, a similar situation may also arise with indirect dependencies. To handle this, the rule uses a notion of codependency. Variables V *and* W *are codependent in* M if there is some variable X such that X depends (perhaps indirectly) on both V and W, and X is modifiable by M. In particular, because V always reflexively depends on V, if V also depends on W, then V and W are codependent in any method in which V is modifiable. In summary, if V and W are codependent in M, then V and W are both modifiable by M.[5]

Additional side-effects rule. Let V be a superclass variable and W a new subclass instance variable. If V and W are codependent in a method M, then M must be overridden, unless the subclass's specification of M prohibits modification of W.

This rule requires a method override if the subclass method can have additional side-effects. A subclass method has *additional side-effects* if it overrides a superclass method and it modifies a concrete instance variable defined in the subclass; such changes are in addition to any that have been specified in the superclass. In class `PointPlusTotal`, any overriding method that modifies `_totalDist_` has additional side-effects.

Formally, an overriding subclass method P may have additional side-effects if

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

The postcondition itself may refer to a new concrete variable W implicitly; that is, it may explicitly reference a variable V that is codependent with W in P. The values of these variables may be related indirectly by either a `represents` clause or an invariant. For example, in Figure 4, the public specification explicitly references `totalDist`; however, it implicitly references the concrete variable `_totalDist_`, through the `depends` and `represents` clauses.

When a subclass method P can modify a concrete variable W, our approach requires both `depends` clauses and an assertion, $A(W)$, implied by P's postcondition. For example, if a subclass method P can modify a model variable V and its specification says it must modify W in order to achieve some effect on V, then a `depends` clause for V is required to allow this [17, 18]. On the other hand, if the `depends` clauses allow P to modify W, then P's postcondition must imply an assertion about the state of W; otherwise, if there is no

[5]In terms of Leino's work [18], V and W are codependent in M if they are in the same data group as a variable that is modifiable in M.

```
package edu.iastate.cs.jml.paper;

public class PointPlusTotal extends Point
{
  //@ public model int totalDist;

  //@ public depends: xCoord -> totalDist;
  //@ public depends: yCoord -> totalDist;
  //@ public depends: oldX -> totalDist;
  //@ public depends: oldY -> totalDist;

  /*@  public_normal_behavior
    @   modifiable: xCoord, yCoord, oldX,
    @               oldY, totalDist;
    @   ensures: xCoord == initX
    @       && yCoord == initY
    @       && oldX == initX && oldY == initY
    @       && totalDist == 0;
    @*/
  public PointPlusTotal(int initX, int initY);

  /*@  public_normal_behavior
    @   ensures: \result == totalDist;
    @*/
  public int getTotalDistance();

  /*@ also
    @  public_normal_behavior
    @   modifiable: totalDist;
    @   ensures: totalDist
    @     == \old(totalDist)
    @        + distance(xCoord - oldX,
    @                   yCoord - oldY);
    @*/
  public void move(int newX, int newY);

  /*@ also
    @  public_normal_behavior
    @   modifiable: totalDist;
    @   ensures: totalDist
    @      == \old(totalDist) + (xCoord - oldX);
    @*/
  public void moveX(int newX);

  /*@ also
    @  public_normal_behavior
    @   modifiable: totalDist;
    @   ensures: totalDist
    @       == \old(totalDist) + (yCoord - oldY);
    @*/
  public void moveY(int newY);
}
```

Figure 4: `PointPlusTotal`'s public specification from the file `PointPlusTotal.jml-refined`.

```
package edu.iastate.cs.jml.paper;

/*@ refine: PointPlusTotal
  @        <- "PointPlusTotal.jml-refined";
  @*/

public class PointPlusTotal extends Point
{
   protected int _totalDist_;

   /*@ protected depends:
     @      totalDist -> _totalDist_;
     @*/
   /*@ protected represents:
     @      totalDist <- _totalDist_;
     @*/
}
```

Figure 5: `PointPlusTotal`**'s protected specification, from** `PointPlusTotal.jml`**.**

```
package edu.iastate.cs.jml.paper;

public class PointPlusInvariant extends Point
{
   /*@  public_normal_behavior
     @   modifiable: xCoord, yCoord,
     @               oldX, oldY;
     @   ensures: xCoord == initX
     @        && yCoord == initY
     @        && oldX == initX
     @        && oldY == initY;
     @*/
   public PointPlusInvariant(int initX, int initY);
}
```

Figure 6: `PointPlusInvariant`**'s public specification, from** `PointPlusInvariant.jml-refined`**.**

assertion, then W can take on an arbitrary value allowed by any represents clause. Thus, another implicit assumption is that the subclass and implementer care about the state of W (in this sense the rule is perhaps too strong for those rare situations when the state of an instance variable does not matter).

If V is declared to depend on W, then all methods in which V is modifiable can also modify W. To avoid having to override the methods that do not need to modify W, one must explicitly specify that each such method does not change W. In JML, this can be done by using the expression `\not_modified(`W`)`. With such a specification, a method without additional side-effects does not need to be overridden based on this rule. (Similar remarks apply to code-dependencies.) In summary, the additional side-effects rule ensures that exactly those subclass methods that may have additional side-effects have been overridden.

In JML, the keyword `invariant` introduces properties that must hold in all publicly-visible states of objects of the class. If more than one public or protected `invariant` clause occurs in a class, then all the given properties must hold in such states. Together, these properties are the *invariant* for the class. For example, subclass `PointPlusInvariant` specified in Figures 6 and 7 has a class invariant, given at the end of Figure 7.

The class invariant must hold at the beginning and end of all public methods. Therefore, when verifying an implementation of a public method, the class invariant will be conjoined with the method's pre- and postconditions. However, because non-public methods are not publicly visible, they may only assume the explicit precondition given in their specification. Furthermore, implementations of non-public methods are only required to establish the explicit postcondition given in their specification.

Due to subclass invariants and represents clauses, the implied assertion, $A(W)$, may not appear explicitly as a conjunct in the postcondition. For example, the invariant in Figure 7 is implicitly conjoined with the postconditions of all public methods of `PointPlusInvariant`. It specifies that the value of concrete variable `deltaDistance` is determined by `deltaX` and `deltaY`. Thus, `deltaDistance` must be allowed to change whenever `deltaX` or `deltaY` changes; this is done using two `depends` clauses, as shown in the protected specification of Figure 7.

The concept of additional side-effects has another important purpose. As will be seen later, downcalls to methods that have additional side-effects can cause problems.

4.2 Temporary Side-Effects

Another problem that can arise when new instance variables are added to a subclass is caused by temporary side-effects. A method has *temporary side-effects* if it modifies an instance variable and then restores the original value before it returns. Temporary side-effects can cause problems if a superclass method makes a downcall before the original value has been restored. For example, consider the implementation of `Point` shown in Figure 8. In that figure, method `distanceTo` is implemented using the `move` and `distanceMoved` methods. This implementation will not work properly when called from subclass `PointPlusTotal`, because `move` has additional side-effects that are not handled. That is, `_totalDist_` will not be restored, in violation of the `modifiable` clause of `distanceTo`. Therefore, `distanceTo` would have to be overridden.

Another problem due to temporary side-effects and downcalls is illustrated by the implementation of `Point` given in Figure 9. In this figure only the method `distanceMoved`, which has no additional side-effects, is used in the implementation of `distanceTo`. However, even this implementation may not work correctly when called from a subclass like `PointPlusInvariant` of Figures 6 and 7, because the subclass invariant is not established before `distanceMoved` is

```
package edu.iastate.cs.jml.paper;

/*@ refine: PointPlusInvariant
  @ <- "PointPlusInvariant.jml-refined";
  @*/

public class PointPlusInvariant extends Point
{
   protected int deltaDistance;

   /*@ protected depends:
     @    deltaX -> deltaDistance;
     @*/
   /*@ protected depends:
     @    deltaY -> deltaDistance;
     @*/

   /*@ protected invariant:
     @    deltaDistance
     @      == distance(deltaX, deltaY);
     @*/
}
```

Figure 7: `PointPlusInvariant`'s protected specification, from `PointPlusInvariant.jml`.

called. Since `distanceTo` might not work correctly, it would have to be overridden.

The first problem described above can be avoided by requiring an override if a superclass method calls a method with additional side-effects. But the second problem cannot be detected using only pre- and postconditions. JML does not allow temporary side-effects[6]; that is, in JML, variables not mentioned in the modifiable clause may not be assigned to by a method (or a method it calls). JML's semantics for the modifiable clause is more restrictive than necessary, but is simpler than our assumption. Recall that we assume that methods do not temporarily change and then restore instance variables around calls to public methods. With this assumption, it suffices to interpret modifiable clauses as only pertaining to modification of the value of variables between the pre- and post-states of a method. However, our assumption is difficult to check statically, whereas the JML rule is easy to check statically.

4.3 Subclass Invariants

Another problem that occurs in the presence of subclass instance variables is related to subclass invariants. For example, a superclass method, M, could modify variables that invalidate a subclass invariant; if this occurs, it is unsafe for M to self-call down to public methods of the subclass since they expect this invariant to hold. The rule below is necessary because it is not possible, without superclass code, to know whether the subclass invariant has been established prior to such calls.

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

```
public class Point {
  protected int x, y, deltaX, deltaY;
  public void move(int newX, int newY){...}
  ...
  public int distanceTo(Point p) {
      int saveX = x, saveY = y;
      int saveDX = deltaX, saveDY = deltaY;
      move(p.x, p.y);
      int d = distanceMoved();
      x = saveX; y = saveY;
      deltaX = saveDX; deltaY = saveDY;
      return d;
  }
}
```

Figure 8: Implementation of class `Point`. The code for `distanceTo` violates our assumptions, because it uses temporary side effects.

```
public class Point {
  protected int x, y, deltaX, deltaY;
  public int distanceMoved() {
      return Math.abs(deltaX)
             + Math.abs(deltaY);
  }
  ...
  public int distanceTo(Point p) {
      int saveDX = deltaX, saveDY = deltaY;
      int d = distanceMoved();
      deltaX = saveDX; deltaY = saveDY;
      return d;
  }
}
```

Figure 9: Another implementation of class `Point`, which also has temporary side-effects.

However, in the statement of the invariant rule, we use the notion of codependency (from Section 4.1) as a way to conservatively detect which variables might be related by an invariant. Recall that if two variables are related by an invariant, and one variable can be changed by a method, then the variables must be codependent in that method, otherwise the other variable could not be changed to maintain the relationship.

Invariant rule. Let S be a subclass of C. Let V be a concrete instance variable of C and let W be a new concrete instance variable of S. Let M be a method of C and O be an argument of M that could be of type S. If V and W are codependent in M and if M can directly or indirectly make a downcall on O to an

overriding public method of the subclass, then M must be overridden.

In a language with a static type system, the static type of the argument O in the invariant rule would be C or some supertype of C. Object O in the invariant rule can only be a subclass object if O is passed explicitly or implicitly to M, since M does not know about subclasses. However, the object O would usually be M's implicit receiver object `this`.

Furthermore, the invariant rule is not concerned with downcalls to non-public methods based on the following reasoning. Only public methods can implicitly depend on class invariants. A non-public method must explicitly state all of its preconditions. Suppose the new subclass invariant is I_S. If a non-public method $C::N$ with precondition Pre is overridden such that the overriding method $S::N$ expects the subclass invariant to also hold, then $S::N$ does not refine $C::N$; that is, $S::N$ has strengthened $C::N$'s precondition to Pre && I_S. Such cases are handled by the method refinement rule of Section 4.5. Therefore, since a superclass method M does not know about subclass invariants, M does not require that these invariants hold unless it directly or indirectly calls down to a public method of the subclass.

Another problem involving subclass invariants is caused by the modification of superclass instance variables by unrelated classes. If some unrelated class modifies instance variables, then the new subclass invariant may no longer hold. This would cause downcall problems that cannot be eliminated by method overrides and is the reason our approach assumes that methods do not access instance variables of objects of unrelated classes.

4.4 Mutually Recursive Methods

This subsection describes an overriding rule that makes termination proofs possible. It prevents potential nontermination that may arise from new methods inserting themselves into cycles of mutually recursive 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.

Callback cycle rule. Let P be an overriding method in a subclass. If P calls a method M that calls back directly or indirectly to P, then M must be overridden.

The callback cycle rule ensures that all methods in a call graph cycle are overridden if any method in the cycle is overridden, since the rule is applied repeatedly until all methods in the cycle have been overridden. 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, in a superclass, a method M may call P which then calls back to M. However, if P is overridden and the new overriding method P does not call M, then this cycle does not occur in the subclass; thus the callback cycle rule would not require that M be overridden.

We assume that groups of mutually recursive methods do not contain methods from unrelated classes. This seems to be needed for soundness because overriding all classes involved in such a cycle does not prevent nontermination. For example, suppose A and C are classes for which no code is available. Further, suppose $A::m$ and $C::n$ are mutually recursive; in this case $A::m$ takes an argument of type C (or accesses a field of type C) and $C::n$ similarly takes an argument of type A. Then, simply by making subclasses B of A and D of C and overriding m in B and n in D does not help the termination proof, because a call to $B::m$ might pass an actual argument of type C, not one of type D, and the code for $C::n$ is not available.[7]

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

1. $Theory(S) \vdash \texttt{\textbackslash old}(Pre_T^m) \Rightarrow \texttt{\textbackslash old}(Pre_S^m)$
2. $Theory(S) \vdash Post_S^m \Rightarrow Post_T^m$
3. $Theory(S) \vdash Modifiable_S^m \subseteq Modifiable_T^m$

Figure 10: A practical notion of method refinement [1, 2, 24, 28].

4.5 Method Refinement

This subsection describes an overriding rule that deals with overriding subclass methods that do not conform to the superclass's specification. If a superclass method calls down to such a non-refining method, it may not behave as expected.

The notion of refinement relates 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, that is, specification B is stronger than specification A [2, 9, 28]. A definition of method refinement that is sufficient for most purposes is given in Figure 10.[8]

In Figure 10, $Theory(S)$ is the set of all formulas that follow from S and the semantics of the OO programming language and specification language. An expression such as $\texttt{\textbackslash old}(P)$ means that predicate P is applied to the pre-state, the program state at the beginning of a method invocation. The first and second proof obligations give the relationship that must hold between the preconditions and postconditions of the methods being overridden by S.[9] The third proof obli-

[7] 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 paper.

[8] This definition can be weakened by replacing proof obligation 2 of Figure 10 with the following [3]:

$$2'.\ Theory(S) \vdash (\texttt{\textbackslash old}(Pre_S^m) \Rightarrow Post_S^m) \Rightarrow (\texttt{\textbackslash old}(Pre_T^m) \Rightarrow Post_T^m)$$

[9] When verifying the implementation of a class, one also

gation says that the variables modifiable by method m in type S must be a subset of the variables modifiable by the overridden supertype method m in T, taking into account any specified dependencies.

Method refinement, in contrast to behavioral subtyping [1, 4, 21, 20], is defined from the point of view of the implementer rather than that of the client. Thus the protected specification must be used in reasoning about method refinement. However, because a subclassing contract does not specify functional behavior, it is ignored in reasoning about method refinement.

An overriding 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.

When $C_{sub} :: P$ is a non-refining method, then its allowed behavior is not a subset of the allowed behavior of the superclass method $C_P :: P$ it is overriding. Therefore, other superclass methods that call down to P may not behave as specified, since they were verified using the superclass specification of $C_P :: P$. A downcall to $C_{sub} :: P$ 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. Hence, the following rule must be considered by the programmer whenever overriding a method.

Method refinement 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.

If a subclass has a non-refining method, then methods must not make a this-argument call to a method that expects a superclass object. Furthermore, in general it is unsound to allow non-refining public methods in subtypes, since this breaks behavioral subtyping [1, 4, 21, 20]. This is why JML enforces method refinement by specification inheritance. However, in languages like C++ one can create a subclass that is not a subtype by using protected or private inheritance. In such cases, one is not concerned with making behavioral subtypes, and hence one will often override a method in a way that makes it non-refining. Thus, this rule is also useful in such languages.

4.6 Concrete Data Refinement

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

conjoins invariants to the pre- and postconditions of public methods. However, for refinement, one does not have to consider invariants; technically this is because one does the proof in the theory of the subclass, where the subclass invariant always holds.

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 [7, 6, 28, 29]. An example of data refinement, occurred above when `Point`'s model variable `oldX` was refined to the concrete variables `x` and `deltaX`.

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 [7, 6, 10, 28, 29]. In our example, this relation was specified by the `depends` and `represents` clauses in the protected specification for `Point` (Figure 2).

We distinguish two kinds of data refinement: model and concrete. Figure 2 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 is that inherited superclass methods would try to refer to missing or changed instance variables.

However, the programmer of a subclass can choose to ignore a superclass variable and use a different data structure in the subclass. Another possibility for concrete data refinement is that the programmer can use the same superclass variables but change the data's properties, and thus the way the data is interpreted and manipulated. This might happen 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. 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), and

2. making data structure changes to improve efficiency in comparison to inherited data structures.

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. "Direct access" means access that names specific instance variables; these may be instance variables of the superclass or instance variables of another object of a related class.

Concrete data refinement rule. A method, M, must be overridden (i) if M makes a direct self-access or object-access to a concrete variable V that is data refined by the new subclass and (ii) if the part of the superclass's invariant that concerns V is not maintained by the subclass.

The above rule is concerned with accesses to variables in objects that are possible subclass objects; these are exactly the accesses that are included in the `accessible` clause (see Section 3.3.2).

The concrete data refinement rule mandates that superclass methods be overridden when they are expecting a different representation. So calling such superclass methods must not be allowed. For example, it would be unsafe to change the way the concrete variable `deltaX` is interpreted and used in the subclass without overriding all methods that access this variable. Such a change of interpretation would mean that the representation invariant of `Point` would no longer hold for subclass objects.

Although we have assumed that unrelated classes cannot access instance variables of a given class, several OO programming languages allow such access. For example, Java allows such access within packages, and C++ allows it with its `friend` feature. Our rule for concrete data refinement is only valid if such accesses are not permitted. 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. However, we did not want to limit our study to JML.

4.7 Super-Calls

In this subsection we discuss a rule for determining when super-calls are safe.

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, if the subclass is not making a concrete data refinement of the entire superclass representation, then an overriding subclass method M must make a super-call to the overridden superclass method $C_M :: M$, if $C_M :: M$ modifies private instance variables or calls private methods with such side-effects. Without the superclass code or more information, it is not possible to know the effect of these calls and accesses on private instance variables. But it is outside the scope of this paper to determine exactly when a super-class call is necessary. We leave that for future work.

However, super-calls are not always safe. Consider the Java implementation, shown in Figure 11, of two of the subclass methods that had to be overridden by `PointPlusTotal` based on the additional side-effects rule. Calling the superclass method and updating `_totalDist_` seems like an obvious implementation. However, the callable clause of method `Point`:: `move` in Figure 3 indicates a self-call to `moveX` and `moveY`. Thus, the variable `_totalDist_` might be updated twice, because `Point`:: `move` calls down to `PointPlusTotal`:: `moveX` (from Figure 11), which also updates this variable. Any superclass method that calls down to a method with additional side-effects cannot safely be called without superclass code.

```
public class PointPlusTotal extends Point {
  protected int _totalDist_;

  ...

  public void move(int newX,int newY) {
      super.move(newX, newY);
      _totalDist_ =
         _totalDist_ + distanceMoved();
  }
  public void moveX(int newX) {
      super.moveX(newX);
      _totalDist_ =
         _totalDist_ + deltaX;
  }
}
```

Figure 11: Incorrect implementation of subclass `PointPlusTotal`.

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. A superclass method is *invalidated by the new subclass* if it might no longer satisfy its superclass specification or calls down to a method that might have additional side-effects (as in the above example). A superclass method might no longer satisfy its superclass specification if it had to be overridden based on one or more of the rules given earlier, aside from the additional side effects rule. That is, a superclass method might no longer satisfy its superclass specification if: (i) it has side-effects that could invalidate a subclass invariant and calls an overridden public method (invariant rule), (ii) it is a member of a callback cycle that had to be overridden (callback cycle rule), (iii) it calls down to a non-refining method (method refinement rule), or (iv) it accesses a variable V and the subclass does not maintain the superclass's invariant concerning V (concrete data refinement rule).

A superclass method that calls down to a method that might have additional side-effects might still satisfy its specification, but without the superclass code (or more information), it is not possible to know what effect calling it may have on subclass instance variables. This was described in the discussion of Figure 11 above. Therefore, a superclass method that calls down to a subclass method with additional side-effects must also be considered to be invalidated.

The 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. Therefore, super-calls to M must

also be prevented.

Super-call authorization. A superclass method may only be called by subclass methods, if it has not been invalidated by that subclass.

The above rule ensures that all methods of the superclass that are super-called satisfy their specifications and have no additional side-effects. A superclass method can only have additional side-effects if it calls a method that has been overridden by a subclass method. For example, in subclass `PointPlusTotal`, the `depends` clauses from Figure 4 and the `modifiable` clauses of methods `move`, `moveX`, and `moveY` from Figure 1 specify that these methods have additional side-effects and have to be overridden; that is, they modify subclass instance variables. Hence, the super-call authorization rule says that any methods in `Point` that directly or indirectly make (down)calls to these methods are invalidated and cannot be called by the subclass. For example, `Point::move` is invalidated because of its self-call (downcall) to `moveX` (Figure 3). Therefore, the super-call in the invalid implementation of `move`, shown in Figure 11, would not be allowed.

On the other hand, the super-call in the implementation of `PointPlusTotal::moveX` shown in Figure 11 is permitted, since `Point::moveX` has not been invalidated by the new subclass.

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 this rule, 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.

4.8 Discussion

In this subsection we discuss two consequences of the rules. The first subsection below describes an overriding rule that deals with unoverridable invalidated methods. The second notes cases in which a correct subclass cannot be written, based on the rules given above.

4.8.1 Unoverridable Methods

From the point of view of a new subclass, an *unoverridable 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 unoverridable because of visibility control; for example, `private` methods are unoverridable in Java and C++; static methods and constructors are also unoverridable. Further, from the point of view of a new subclass, S, methods in classes that are unrelated to S are also unoverridable. The following rule deals with invalidated, non-public methods that are unoverridable; its purpose is to prevent such methods from being invoked.

Unoverridable method rule. If a non-public superclass method, 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 called by subclass methods.

Notice that the above rule only applies to non-public methods. If M were public, then behavioral subtyping would fail for the subclass, because M, an invalidated, non-overridable method, could be called by clients of the subclass.

Furthermore, in a language like C++, one can use protected inheritance without worrying about behavioral subtyping. However, if one inherits an unoverridable, invalidated public method, there is no way to prevent clients from calling it. Thus there would be no way to guarantee that such a method meets its specification.

4.8.2 Unimplementable Subclasses

An interesting result of our study is that in some situations it may not be possible to write a verifiably correct implementation of a subclass specification without seeing the superclass's code. One such situation is when, based on the rules, a public method must be overridden, and that method is unoverridable, or when a method in an unrelated class is invalidated by the new subclass.

Another example occurs in situations like the following. Suppose part of the superclass object state is private and is maintained by a non-public method M. If M calls a method that may have additional side-effects, then a provably correct implementation satisfying the class specification cannot be created without some trick like saving and restoring the object state, which our technique prohibits.

One way to avoid these problems is to allow all methods in a class to be overridable (e.g., avoiding the use of `final` in Java), and to allow subclasses to directly access all instance variables (e.g., avoiding the use of `private` in Java). We leave for future work a classification of such situations and ways to avoid them.

5. SOUNDNESS

The soundness of our 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. We assume the existence of a verification logic for the OO programming and specification languages (e.g., [31]) and that the superclass code has been verified as correct. We assume the new subclass has a formal specification with the three parts described above.

The partial correctness proof for the subclass would be done in the standard way using method specifications for method calls; that is, one would use the superclass specifications for both super-calls and self-calls to inherited methods, and the subclass specifications for self-calls to subclass methods. Object-calls are handled using the specification of the static

type of the receiver [16]. The proof of termination would also be standard if the code for all methods in a mutually recursive callback cycle is available.

To have a valid partial correctness proof, all superclass methods called by the subclass must satisfy their superclass specification and must not have additional side-effects; that is, a subclass must not call invalidated (see Section 4.7) superclass methods. We also assume that a subclass does not have non-refining, public methods, since in general such subclasses can have downcall problems that cannot be prevented via method overrides. We assume that these are the only ways that a subclass can invalidate superclass methods; proving this assumption is outside the scope of this paper and would require choosing a programming language and a specification language and doing a formal analysis of this combination. We leave that as future work. However, this assumption seems reasonable based on our assumptions in Section 2. The following theorems ensure that a standard proof will be valid.

Valid Specification Theorem: If none of the rules given in this paper are violated in the implementation of subclass methods, then the subclass code does not call any invalidated superclass methods.

Proof sketch: Induction on the number of overridden methods using the rules and the above assumption about the only ways to invalidate superclass methods.

Mutual Recursion Theorem: If none of the rules given in this paper are violated in the implementation of subclass methods, then there are no callback cycles involving both subclass and superclass methods.

Proof sketch: The callback cycle rule and the super-call authorization rule prohibit callback cycles involving both subclass and superclass methods.

The valid specification theorem means that superclass methods called by the subclass satisfy their specifications. Furthermore, the mutual recursion theorem means that the code needed for a termination proof of mutually recursive methods is available. Thus, if the code of a new subclass follows the rules, then a standard proof of its correctness would be valid.

6. TOOL SUPPORT

Tool support is necessary to automatically create subclassing contracts and to assist in applying the rules. This section describes our plans for a tool that will fill this role for JML and Java.

Our proposed tool will have three phases. It will work on a per-class basis, looking at related classes mentioned in the code. For a given class, the first phase will typecheck the code and generate its subclassing contract (i.e., the callable and accessible clauses).

The subclassing contract always reflects the current calling structure of a class no matter how many layers there are in the class hierarchy; that is, the subclassing contract of a method defined in the subclass is derived directly from the code of that method, but the subclassing contracts of inherited methods are inherited in the same way that methods are inherited by subclasses.

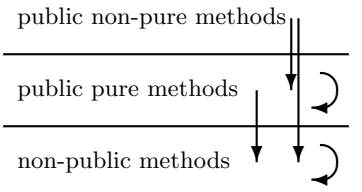

Figure 12: The three levels of methods used in the guidelines for library providers. Arrows indicate that calls can be made in the arrow's direction.

Phase 2 will generate the transitive closure of the callable clauses to obtain the methods directly or indirectly called. From this the directly and indirectly called methods can be listed so the invariant rule can be applied more easily. The details of how the transitive closure will be computed is given in Appendix A.

Phase 3 will check to make sure, based on our rules, that all methods have been properly overridden. For example, the tool may say that a superclass method should be overridden. The tool may also complain that a method was not overridden properly if the overriding method makes a super-call to an invalidated superclass method.

7. CLASS LIBRARIES

A major purpose of our study was to identify the kinds of problems that arise when creating a subclass without the superclass code. For each potential problem identified, rules or restrictions were given that prevent such problems. The rules in our paper are conservative, but, based on our experimentation with much more complicated examples, we believe that these rules identify issues that must be considered by OO programmers in general when creating subclasses. Even if the superclass code is available, the calling structure and fields accessed must be considered when determining which methods to override.

However, our study also provides some insights for framework and class library designers and implementers. The idea is to use three levels of methods: public non-pure, public pure, and non-public (see Figure 12). The visibility of the methods in each level is given by the name; for example, public non-pure methods would be declared `public` in Java. Public non-pure methods cannot be called by other methods, but may call methods in the other two levels. The public pure methods can only be called by public methods; they may call pure public and non-public methods. The non-public methods can be freely called by all other methods of the class, and may only call other non-public methods.

The idea described above is the main one embodied in the following guidelines. To these we add a few other assumptions and details, for soundness. If the library provider and reuser adhere to these guidelines, then reasoning about creating valid subclasses is simplified. These guidelines are overly restrictive in some cases, but they guarantee that a reuser can easily make a correct subclass without seeing the library's source code. By "easily" we mean that the reuser only needs to follow the simple guidelines for programmers described later in this section; that is, the reuser does not need to have read this paper and does not need to think about the application of our rules. In particular, these two sets of guidelines guarantee that methods in the library are never invalidated, and so super-calls are always safe.

7.1 Guidelines for Implementers of Libraries

For each library class:

L1 methods should not directly access any instance variables from unrelated classes,

L2 methods should not call the public methods of an object while temporary side-effects have not been restored in that object, and

L3 overriding methods should refine the method being overridden.

For each library class that can be subclassed by reusers:[10]

L4 there should be no mutually-recursive methods,

L5 methods should not make this-argument calls,

L6 methods should not self-call or super-call non-pure public methods,

L7 non-public methods should not call the public methods of a related class,

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 protected invariant.

Guidelines L1 and L2 are assumptions made in our approach and are necessary for its soundness; as explained earlier, they prevent problems related to calls while a subclass invariant does not hold. Note that guideline L1 holds automatically in Smalltalk, where there are no public instance variables. Guideline L3 is fundamental to behavioral subtyping; it is built-in to JML.

Guideline L4 eliminates mutual recursion, and thus the need for reusers to think about the callback cycle rule. Guideline L5 eliminates problems caused by callbacks. This guideline disallows mutual recursion among unrelated classes and prevents problems related to subclass invariants involving this-argument calls.

[10] A library class cannot be extended by a reuser if it is private or `final` in the Java sense.

Guideline L6 gives reusers the ability to prevent non-public superclass methods from having additional side-effects. The only way a superclass method can have additional side-effects is if it calls down to a subclass method with additional side-effects. Guideline L6, if followed, means that superclass methods must directly modify instance variables or call non-public methods to implement side-effects. This guideline means that the only methods with side-effects that can be called are non-public methods. Since non-public methods are not required to establish invariants, they are also not required to have additional side-effects. This allows reusers to avoid invalidation of superclass methods caused by additional side-effects; they can do this by not overriding these non-public methods.

Guidelines L2, L5, and L7 mean that non-public methods cannot be invalidated by subclass invariants. Thus, in all cases, superclass methods are not invalidated and can be called. Guideline L8 and the others make it possible to avoid concrete data refinement, which we have assumed does not occur.

7.2 Guidelines for Reusers Inheriting from Libraries

Programmers who are using libraries or frameworks that follow the above guidelines, but are not themselves producing extensible libraries or frameworks, do not need to follow the above guidelines. However, reusers should pay attention to the following guidelines:

R1 avoid additional side-effects when overriding non-public methods that modify superclass instance variables,

R2 avoid creating a group of mutually recursive methods involving a method of a library superclass,

R3 when overriding a superclass method, always refine it,

R4 make the subclass's protected invariant imply the superclass's protected invariant, when superclass instance variables have been exposed to the subclass (or other classes).

If the above guidelines are followed, then super-calls will be safe. Note that JML requires that guidelines R3 and R4 are followed.

7.3 Discussion

Following both sets of guidelines ensures that additional side-effects do not invalidate other methods. This means that methods can be implemented and reasoned about independently of each other.

Our guidelines could also be used to help with the organization of a class library or framework. Its implementation could be reviewed for its reusability using the subclassing contracts of its classes in light of the guidelines for library providers.

We leave as future work making these guidelines less restrictive for library implementers. Another direction of future

work might be further restrictions that would make fewer demands on reusers.

7.4 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 [19, 24, 25] is essential. For example, guideline L8 says that libraries have to be documented with public and protected invariants, and this is required by guidelines R3 and R4 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 modifiable clauses to help make them more understandable and readable [19]. 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 reusers writing subclasses.

Guidelines L6 and L7, illustrated in Figure 12, divide the methods of a library class into layers. The layers themselves are a substitute for the subclassing contract. That is, they eliminate the need for library documentation to include something like the subclassing contract. However, if a library provider cannot strictly follow this layering, something like the callable and accessible clauses in the subclassing 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 guidelines R1 and R2.

8. RELATED WORK

Leino introduces the notion of data groups and dependencies for controlling which subclass fields can be modified by an overriding subclass method [17, 18]. The `depends` clause in JML is derived from Leino's work. A specification language needs a feature like the `depends` clause to allow a tool to apply the additional side-effects rule. 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" [12]. 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 [12, 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 [12, sections 4.5 and 4.6]. Thus, a group would 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 [13]. 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 or if they have additional side-effects.

Steyaert, Lucas, *et al.* introduce a similar approach called "reuse contracts" for specifying a contract between a class and its subclasses [22, 38]. 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, although 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.

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 [33] 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.

Stata and Guttag [36] solve this semantic problem by requiring that new subclasses implement behavioral subtypes [1, 4, 21, 20] of their superclass. They extend the partitioning

ideas of Kiczales and Lamping [12] and Lamping [13] 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 [5]. 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 [35]. 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 mutually recursive methods. 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 [26]. 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 [13], Stata and Guttag [36] and Steyaert, Lucas, *et al.* [38]. 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 non-overridable. 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* [11], whereas subclassing 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 subclassing contract, a major part of our approach, would be generated automatically by our proposed tool. Having a tool to make sure no rules are violated helps ease the work of applying the rules and thus automates part of the work involved in creating correct subclasses. In addition, except for Stata [35], 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 additional side-effects.

Szyperski shows how downcall and callback problems are avoided by using object composition and message forwarding rather than implementation inheritance [39, pp. 117-119]. 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. In some sense, forwarded calls are like super-calls except that they 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 contains the original object). Object composition would have worked fairly well in the design of class `PointPlusTotal`, except that protected methods would not have been 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 others are difficult and complicated [39, p.118].

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 discipling inheritance to avoid such problems [27]. Their requirements prohibit access to superclass instance variables and do not allow instance variables to be declared in subclasses. Our technique, which relies on the extra information contained

in the `accessible` clause, allows more flexible access to instance variables. 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, this does not handle problems caused by additional side-effects. Their technique does not permit additional side-effects, but our rules allow sound reasoning about the state of subclass instance variables and additional side-effects.

Changes in the subclassing 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 but that method was already defined in an existing subclass [22, 38]. 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.

In summary, our approach has important advantages over all the work described above in that it allows super-calls and reasoning about when such calls are safe. Furthermore, creation of the subclassing contract need not be part of the manual design activities of a class library.

9. CONCLUSION

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 in 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 subclassing contract that provides important additional information needed by programmers to safely specialize or extend a class.

The subclassing 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, some programmers may worry that they are revealing too much detail about their methods. However, we believe that the subclassing 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 [19, 25]. Both parties benefit because the specification, and in particular the subclassing 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 subclassing contract of the old version of a class is broken by the new version. A subclassing contract is *broken* if it has additional accesses or calls that do not appear in the old version. A broken subclassing 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. However, it remains future work to carefully analyze programming languages to determine whether these are the only ways in which downcall problems can arise. Doing this would help establish the soundness of our technique for subsets of realistic languages.

Our approach to creating subclasses is general in the sense that it does not impose restrictions (other than those given in Section 2) 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 Section 7). These guidelines place greater demands on the library implementer, but make only limited demands on the reuser.

Our work also points out some directions for future work on JML. Although JML does not provide a way to list the methods that may be indirectly called, one can imagine adding to the subclassing contract a `callable_indirectly` clause that would list the indirectly called methods. Furthermore, one could add a `recursive_methods` clause, listing the methods in a group of (two or more) mutually recursive methods. This could also be generated by the tool. Recording these additions in the subclassing contract would assist people in manually applying the invariant and callback cycle rules.

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 [23]. Another area is to relate concrete data refinement to refactoring [32]. Longer term future work includes justifying the assumptions about the programming language we used in arguing the soundness of our technique.

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 subclassing contract of the new subclass, and check for violations of the rules. Building this tool is important future work because it can catch and help prevent potential errors prior to execution.

APPENDIX

A. COMPUTING THE TRANSITIVE CLOSURE

The tool will compute the transitive closure of the `callable` clause for each method. When computing the transitive closure, `callable` clauses of a subclass C_{sub} must be interpreted from the point of view of a receiver object of type C_{sub}. For example, if C_{sub} overrides a method P, then self-calls to P appearing in C_{sub}'s `callable` clauses are interpreted as calls to $C_{sub} :: P$. However, if C_{sub} inherits P from a superclass C_P, then such self-calls to P are interpreted as calls to $C_P :: P$. Therefore, when computing the transitive closure of a method, calls will be represented as $(C_M :: M,\ C_R)$, where $C_M :: M$ is the method or constructor called and C_R is the type of the receiver.

Let C_{sub} be a new subclass. Then the set of methods and constructors, TC_M, transitively called by $C_M :: M$ can be computed starting from the set containing the single element $(C_M :: M,\ C_{sub})$ by recursively adding method calls, as described below, until TC_M reaches a fixed point.

For each $(C_N :: N,\ C_R)$ in TC_M and for each call to P in $C_N :: N$'s `callable` clause, do one of the following:

- if P is a self-call, then
  add $(C_P :: P,\ C_R)$ to TC_M, where C_P is the class in which P is defined from the point of view of a receiver of type C_R;
- if P is a super-call that is not a superclass constructor call, then
  add $(C_P :: P,\ C_R)$ to TC_M, where C_P is the class in which P is defined as specified in the super-call (e.g., the superclass of C_N in Java, the named class in C++);
- if P is a superclass constructor call (e.g., `super`$(\ldots)$[11] in Java), then
  add $(C_{\text{new}} :: C_{\text{new}}(\ldots),\ C_R)$ to TC_M, where C_{new} is the class in which the superclass constructor $C_{\text{new}}(\ldots)$ is defined;
- if P is a subclass object-call, then
  add $(C_P :: P,\ C_{sub})$ to TC_M, where C_P is the class in which P is defined from the point of view of C_{sub} (i.e., always assume the receiver has type C_{sub});
- if P is a new object constructor call of the form "`new` $C_{\text{new}}(\ldots)$," then
  add $(C_{\text{new}} :: C_{\text{new}}(\ldots),\ C_{\text{new}})$ to TC_M (note that the type of the receiver may change).

Acknowledgements

Thanks to Raymie Stata and Curt Clifton for helpful comments on earlier drafts. Thanks to the OOPSLA 2000 referees for helpful comments on an earlier version of this paper.

B. REFERENCES

[1] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, 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.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[3] Y. Chen and B. H. C. Cheng. A semantic foundation for specification matching. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, New York, NY, 2000.

[4] K. K. Dhara and G. T. Leavens. 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, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[11] $(\ldots)$ represents the list of types of the arguments that are used to disambiguate overloaded methods or constructors.

[5] S. H. Edwards. Representation inheritance: A safe form of "white box" code inheritance. In *Fourth International Conference on Software Reuse*, pages 195–204. IEEE Computer Society Press, Apr. 1996.

[6] P. H. B. Gardier and C. Morgan. A single complete rule for data refinement. In Morgan and Vickers [30], pages 111–126.

[7] P. H. B. Gardiner and C. Morgan. Data refinement of predicate transformers. In Morgan and Vickers [30], pages 71–84.

[8] J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[9] E. C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[10] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[11] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Mass., 1991.

[12] G. Kiczales and J. Lamping. Issues in the design and documentation of class libraries. *ACM SIGPLAN Notices*, 27(10):435–451, Oct. 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[13] J. Lamping. Typing the specialization interface. *ACM SIGPLAN Notices*, 28(10):201–214, Oct. 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[14] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, Feb. 2000. See `www.cs.iastate.edu/~leavens/JML.html`.

[16] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

[17] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[18] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, Oct. 1998.

[19] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[20] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[21] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, Oct. 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[22] C. Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, Sept. 1997.

[23] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA programming Language*. Addison-Wesley Inc, 1993.

[24] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, 1988.

[25] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[26] M. Mezini. Maintaining the consistency and behavior of class libraries during their evolution. *ACM SIGPLAN Notices*, 32(10):1–21, Oct. 1997. *Conference Proceedings of OOPSLA '97*.

[27] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, 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.

[28] C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[29] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, May 1990.

[30] C. Morgan and T. Vickers, editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, NY, 1994.

[31] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.

[32] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[33] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, Jan/Feb 1990.

[34] A. D. Raghavan. Design of a JML documentation generator. Technical Report 00-12, Iowa State University, Department of Computer Science, July 2000.

[35] R. Stata. 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.

[36] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. *ACM SIGPLAN Notices*, 30(10):200–214, Oct. 1995. *Proceedings of OOPSLA '95 Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications.*

[37] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[38] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA '96 Conference on Object-Oriented Programming Systems, Languagges and Applications*, pages 268–285. ACM Press, October 1996. ACM SIGPLAN Notices, Volume 31, Number 10.

[39] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 1998.