JML's Rich, Inherited Specifications for Behavioral Subtypes

Gary T. Leavens

TR #06-22
August 11, 2006

Keywords: Supertype abstraction, behavioral subtype, behavioral subtyping, modularity, specification inheritance, method specification refinement, join of method specifications, also, specification case, invariants, JML, Java.

2006 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques — Object-oriented design methods; 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; D.3.1 [*Programming Languages*] Formal Definitions and Theory — Semantics; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs and Features — classes and objects, 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.

This is a preprint of an Invited talk to appear in Zhiming Liu and He Jifeng (eds), *Proceedings, International Conference on Formal Engineering Methods (ICFEM'06), Macao, China*, pages 2-36. Volume 4260 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006

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

JML's Rich, Inherited Specifications for Behavioral Subtypes*

Gary T. Leavens

Department of Computer Science, Iowa State University
229 Atanasoff Hall, Ames, IA 50011-1041, USA
leavens@cs.iastate.edu

Abstract. The Java Modeling Language (JML) is used to specify detailed designs for Java classes and interfaces. It has a particularly rich set of features for specifying methods. This paper describes those features, with particular emphasis on the features related to specification inheritance. It shows how specification inheritance in JML forces behavioral subtyping, through a discussion of semantics and examples. It also describes a notion of modular reasoning based on static type information, supertype abstraction, which is made valid in JML by methodological restrictions on invariants, history constraints, and initially clauses and by behavioral subtyping.

1 Introduction

Work on formal methods is interesting for at least two reasons: it can lead to practical tools (such as runtime assertion checkers or model checkers) and it can be used to give insight into informal programming practice. Both of these reasons drive the work on the Java Modeling Language, JML [10, 12, 44, 45, 47]. While JML is technically limited to precisely describing the syntactic interfaces and functional behavior of sequential Java classes and interfaces at the detailed design level, its rich set of specification constructs can be used to explain concepts that can be informally applied in other settings. This paper attempts to give such an explanation for ideas related to behavioral subtyping.

1.1 Context

JML builds on the ideas of Eiffel [58, 59] and the Larch family of behavioral interface specification languages [17, 30, 39, 81]. While JML also uses ideas from other sources [8, 35, 48, 49, 61, 62, 67, 71, 79], at its core it blends these two language traditions.

From Eiffel JML takes the idea of writing assertions (e.g., pre-and postconditions) in program notation; hence JML assertions are written as Java expressions. This helps make JML easy to read by Java programmers.

From Larch JML takes the idea of using mathematical values to specify complete functional behavior. These mathematical values are specified in JML as the values of "model fields" [18, 48]. (Examples of model fields are given below.) Because model fields are only used in assertions, their time and space efficiency is not important. (One

* This work was supported by NSF grant CCF-0429567.

can always turn off assertion checking to gain efficiency.) Thus the type of a model field can be something closer to mathematics, such as a set, sequence, or relation, instead of a binary search tree, an array, or a hash table. This allows users to focus on clarity.

JML has been successful in attracting researchers working with formal methods at the level of detailed design for Java. To date there are at least 19 groups doing research with JML. (See `http://www.jmlspecs.org/` for a list.)

1.2 JML's Tools

Part of JML's attraction for researchers is that they can build on a rich language and use several different tools [10]. This set of tools helps researchers be more productive. For example, since program verification with a theorem prover is time-consuming, it helps to use other tools to find bugs first.

While JML's tools could use more polish, they have been used in several college classes and in work on Java smart cards [9, 10, 11, 33, 57].

The JML research community is dedicated to finding ways to help make the cost of writing specifications worthwhile for its users by providing added value through tools. This includes basic tools such as a documentation generator (jmldoc), and a runtime assertion checker (jmlc) [14, 16]. There are also several tools designed to do formal verification with interactive theorem provers, including LOOP [33, 34], JACK [9], KRAKATOA [57], Jive [73], and KeY [1]. One of JML's principal design goals is to support the use of both runtime assertion checking and formal verification with theorem provers. But JML is also supported by several other novel tools, including:

- ESC/Java2 [36], a descendant of the extended static checker ESC/Java [29], which statically detects bugs and uses specifications to improve the accuracy of bug reports,
- Daikon [27], a tool that mines execution traces to find likely program invariants, which can synthesize specifications,
- The jmlunit tool [15], which uses JML specifications to decide the success or failure of unit tests, and
- SpEx-JML [77], which uses the model checker Bogor [76] to check properties of JML specifications.

Most of these tools are open source products. The JML community itself is also open and encourages more participation from the formal methods community.

1.3 Overview

JML has evolved from its roots in Larch and Eiffel into a language with a rich set of features. The goal of this paper is to use a subset of these features to help explain the ideas connected with behavioral subtyping in a way that will help readers apply them in programming practice, e.g., in documenting and informally reasoning about object-oriented programs. A secondary goal is to aid readers who would like to use some of the JML tools or would like to reuse or build on the ideas of JML's language design. This paper assumes the reader is familiar with basic concepts in formal methods, such as first-order logic and pre- and postconditions [25, 30, 31, 59, 60].

This paper focuses on behavioral subtyping [2, 3, 4, 23, 28, 38, 41, 42, 43, 55, 59, 74] for several reasons. The first is because of its utility in organizing and reasoning about object-oriented software [19, 37, 43, 53, 59]. The second is that JML embodies a set of features that make working with behavioral subtyping particularly convenient, but these features and their combination in JML have not previously had a focused explanation. While the key language design ideas of specification cases [80] and their use in specification inheritance [79] to force behavioral subtyping have been explained in the context of Larch/C++ [23, 40], they are found in a simpler and thus more understandable form in JML. And while these ideas have been previously explained from a theoretical perspective [42], their embodiment in JML makes the explanation more concrete and accessible to the practice of programming and specification language design.

2 Background: JML Specifications for Methods and Types

This section gives background on JML and several examples. This background is necessary to explain the concept of supertype abstraction in the context of JML, since supertype abstraction involves reasoning about specifications. The examples in this section will also be used in the remainder of the paper.

2.1 JML Basics

For example, take the Java interface `Gendered` given in Fig. 1. `Gendered`'s behavior is specified in its JML annotations.[1]

```
public interface Gendered {
  //@ model instance String gender;

  //@ ensures \result <==> gender.equals("female");
  /*@ pure @*/ boolean isFemale();
}
```

Fig. 1. A JML specification of the interface `Gendered`. The JML annotations are written in comments that start with an at-sign (`@`). The rest of the JML notation is explained in the text.

The second line of the figure is an annotation that declares a field `gender`. In that declaration, the modifier **`model`** says that the field is a specification-only field that is an abstraction of some concrete state [18, 48]. The modifier **`instance`** means that this abstraction is based on instance fields, and thus this modeling feature can be thought of as a field in each object that implements the `Gendered` interface.

In JML, specifications for methods precede the header of the method being specified. In Fig. 1, the **`ensures`** clause, specifies the postcondition of the method `isFemale`.

[1] JML annotations should not be confused with Java 5's annotations, which are quite different.

This postcondition says that the value returned by the method, `\result`, is equivalent (written `<==>`) to whether the model field `gender` equals the string `"female"`. The `isFemale` method is also specified using the modifier **pure**, which says that the method cannot have side effects and may thus be used in assertions.

2.2 Specification for fields

The class `Animal` in Fig. 2 will be used to explain JML's features related to fields. Since `Animal` is a subtype of `Gendered`, it inherits the model instance field `gender` (declared in Fig. 1). Inheritance of instance fields means that specifications for instance methods written in supertypes make sense when interpreted in their subtypes. For example, the ensures clause of the method `isFemale` specified in the interface `Gendered` makes sense in its subtype `Animal`.[2]

```
public class Animal implements Gendered {
  protected boolean gen; //@ in gender;
  //@ protected represents gender <- (gen ? "female" : "male");

  protected /*@ spec_public @*/ int age = 0;

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender;
  //@ ensures gender.equals(g);
  public Animal(final String g) { gen = g.equals("female"); }

  public /*@ pure @*/ boolean isFemale() { return gen; }

  /*@   requires 0 <= a && a <= 150;
    @   assignable age;
    @   ensures age == a;
    @ also
    @   requires a < 0;
    @   assignable age;
    @   ensures age == \old(age);   @*/
  public void setAge(final int a) { if (0 <= a) { age = a; } }
}
```

Fig. 2. Class `Animal` from the file `Animal.java`. In a multi-line annotation at-signs at the beginnings of lines are ignored. The other new JML features are explained in the text.

[2] Inherited specifications make sense even if there are shadowing field declarations in subtypes. However, in this paper I will assume that there is no such field shadowing, as this simplification does not lose any generality.

The **in** clause, which occurs immediately after the declaration of the protected boolean field gen, is used to declare datagroup membership. It says that gen is in gender's data group [49]. The data group of a field f can be thought of as a set of fields that are allowed to be assigned to when f is mentioned in an assignable clause. The data group of a model field f includes all the fields needed to determine f's value, but may also include other fields. Thus the **in** clause in the declaration of gen tells JML that: (a) the value of gender may depend on the value of gen and (b) gen may be assigned whenever gender is allowed to be assigned by a method. For example the constructor's **assignable** clause lists gender, which means that it may assign to all locations in gender's data group, which includes gen. (See below for more about assignable clauses.)

The **represents** clause gives an expression for the value of the model field gender. Thus, whenever gender occurs in a specification, such as in the constructor's postcondition, its value is the value of the expression `(gen ? "female": "male")`. However, not only is gender more concise, it is public, whereas gen and the design decision about how gender is represented are hidden from clients. This illustrates how model fields in JML can be used to hide design details [18, 48]. The represents clause specifies an abstraction function [32] from part of the concrete state of an Animal object to a model field. Since the model field gender is inherited from Gendered, this abstraction function can be thought of as mapping part of the state of an Animal object to the state of a Gendered object [55].

The **spec_public** modifier in the declaration of age can be thought of as shorthand for the declaration of a public model field (named age), and clauses saying that the protected field (renamed to, say, _age) is in the model field's data group, and that the model field's value is the value of the concrete field. Use of **spec_public** is often convenient when documenting existing code. It allows the protected field that is used in the representation to be changed (e.g., renamed) at a later date without affecting the specification's clients. If such a change is made in the future, at that time one has to unpack these shorthands and rename all uses of the protected field.

The **requires** clauses in the constructor and method specifications of Fig. 2 specify preconditions. For example, the precondition of the constructor says that the argument g must be either `"female"` or `"male"`.

An **assignable** clause gives a frame axiom [6, 62]. It lists the fields whose data groups may be assigned during the execution of the method. All locations that are not in the data group of a listed field are not allowed to be even temporarily changed. (In this sense JML's assignable clauses are more strict than the modifies clauses found in Larch.) Such frame axioms are important for formal verification [6, 12]. An assignable clause can be thought of as syntactic sugar for part of the method's postcondition. For example the assignable clause of the constructor can be thought of as shorthand for adding **\only_assigned**(gender) to the constructor's postcondition. The method modifier **pure** is, in part, a shorthand for the clause **assignable \nothing** and hence can also be thought of as shorthand for part of a postcondition.

2.3 Joining Specification Cases with `also`

The `setAge` method in Fig. 2 on page 5 has a specification with two specification cases connected by **`also`**. A JML *specification case* consists of several clauses, including requires, assignable, and ensures clauses [47]. Each specification case has a precondition (which might default to true), that tells when that specification case applies to a call. JML's **`also`** joins together specification cases in a way that makes sure that, whenever a specification case's precondition holds for a call, its postcondition must also hold. That is, in general a JML method specification may consist of several specification cases, and all these specification cases must be satisfied by a correct implementation.

One reason for using **`also`** and separate specification cases is to make distinct execution scenarios clear to the specification's reader. In the `setAge` example, a call that satisfies the first specification case's precondition must set `age` to the value of the argument `a`. The second specification case describes the method's behavior for negative arguments. In this case the value of the `age` field must be unchanged. This is specified with the postcondition `age == \old(age)`, which says that the post-state value of `age` must equal its pre-state value, `\old(age)`. The `\old()` operator is often used in the postconditions for methods that change the state of an object [59].

I will refer to the combination of two method specifications with **`also`** as their "join," since it is technically the join with respect to the refinement ordering on method specifications [42, 50, 64]. (It is also easier to talk about "joining" specification cases.)

To define the join operation precisely I will use a bit of notation. As we have seen, specification cases are essentially pairs of pre- and postconditions (the assignable clause being shorthand for part of a postcondition, as explained above). So, in what follows, I will write $T \triangleright (\mathit{pre}, \mathit{post})$ for a specification case of an instance method that type checks when its receiver (**`this`**) has static type T. Thus you can think of $T \triangleright \mathit{spec}$ as being written in type T. In JML, $T \triangleright \mathit{spec}$ will also type check in a context where **`this`** has some subtype of T. I omit the receiver's type when it is clear from context. Also, since there is little difference between a simple method specification and a specification case, I will often just call them method specifications. With this notation, the definition of the join operation for specification cases is as follows [23, 40, 42, 50, 64, 80].

Definition 1 (Join of JML method specifications, $\sqcup^U$). *Let $T' \triangleright (\mathit{pre}', \mathit{post}')$ and $T \triangleright (\mathit{pre}, \mathit{post})$ be specifications of an instance method m. Let U be a subtype of both T' and T. Then the* join of $(\mathit{pre}', \mathit{post}')$ and $(\mathit{pre}, \mathit{post})$ for U*, written $(\mathit{pre}', \mathit{post}') \sqcup^U (\mathit{pre}, \mathit{post})$, is the specification $U \triangleright (p, q)$ with precondition p:*

$$\mathit{pre}' \;\texttt{||}\; \mathit{pre} \tag{1}$$

and postcondition q:

$$(\texttt{\textbackslash old}(\mathit{pre}') \texttt{ ==> } \mathit{post}') \texttt{ \&\& } (\texttt{\textbackslash old}(\mathit{pre}) \texttt{ ==> } \mathit{post}). \tag{2}$$

In the above definition, the precondition of the join of two method specifications is their disjunction (with `||` as in Java). The postcondition of the join is a conjunction of implications (written `==>` in JML's notation), which says that when one of the preconditions holds (in the pre-state), then the corresponding postcondition must hold.

The ability to join method specification cases is useful in specification inheritance, which joins specification cases from subtypes with those inherited from supertypes. However, when the join's receiver type is clear from context, I omit the superscript U.

For example, the join of the two specification cases for `setAge` in Fig. 2 on page 5 is equivalent to the specification case shown in Fig. 3. Of course, one could write this specification directly, but when one compares it to the specification of `setAge` in Fig. 2, one can see that the postcondition of Fig. 3 contains within it a repetition of the preconditions from Fig. 2. This repetition is a maintenance problem and distracts from the clarity of the specification. JML's **`also`** avoids these problems.

```
//@   requires (0 <= a && a <= 150) || a < 0;
//@   assignable age;
/*@   ensures (\old(0<=a && a<=150) ==> age==a)
  @        && (\old(a<0) ==> age==\old(age));   @*/
public void setAge(final int a);
```

Fig. 3. The join of the specification cases for the `setAge` method from Fig. 2 on page 5.

2.3.1 Using `\same` to make refinements Often in writing a method specification in a subtype, one wants the precondition of the overriding method to be the same as that of the specification of the method being overridden. This often occurs for a method m in a subclass that calls `super.`m and then does something extra. JML's predicate **`\same`** can be used in the precondition of such a specification to say that the method's precondition is the same as that of the method being overridden [47]. For example, in Fig. 4 on the next page, the precondition of the given specification case for `setAge` is equal to that in the specification of `setAge` in `Animal`. In this example, that precondition is equivalent to the disjunction of `setAge`'s preconditions from the two specification cases in Figure 2 (as shown in Fig. 3), and is thus is equivalent to `a <= 150`.

2.4 Invariants, History Constraints, and Initially Clauses

In addition to field declarations and method specifications, a type specification in JML may also contain invariants, history constraints, and **`initially`** clauses.[3] An invariant [32] is a predicate that should hold in all *visible states*, i.e., in the pre-state and post-state of each (non-helper[4]) method execution [47, 63], and in the post-state of each constructor execution. Invariants are one-state predicates; i.e., they cannot use **`\old()`**. By contrast a history constraint [55, 56] is a two-state predicate that uses **`\old()`** to state a monotonic relationship between pre-states and post-states. A history constraint

[3] This list is a simplification, but it covers the most important features.

[4] In JML a private method or constructor can be declared with the modifier **`helper`**. This exempts it from having to preserve invariants, or establish history constraints and initially clauses.

```
public class Person extends Animal {
  protected /*@ spec_public @*/ boolean ageDiscount = false; //@ in age;

  /*@ also
    @   requires \same;
    @   assignable age, ageDiscount;
    @   ensures 65 <= age ==> ageDiscount;   @*/
  public void setAge(final int a) {
    super.setAge(a);
    if (65 <= age) { ageDiscount = true; }
  }

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender;
  //@ ensures gender.equals(g);
  public Person(final String g) { super(g); }
}
```

Fig. 4. A JML specification of the class `Person`. The notation `\same` is explained in the text. In JML, `also` must be used in a method specification whenever one overrides a method, to remind the specification's reader about specification inheritance, as will be explained later.

must hold in the post-state of every (non-helper) method execution [47]. An initially clause [26] is a predicate that should hold in all post-states of (non-helper) constructors. Initially clauses are one-state predicates.

In JML all of these clauses may be omitted (as in the examples given previously), in which case a default predicate, `true`, is used. These defaults allow us to speak of "the invariant" etc. declared by a type, even if none is explicitly declared.

2.4.1 Invariants To explain invariants in JML, consider Fig. 5 on the following page. This figure has two `invariant` clauses, both of which declare public (client-visible) instance invariants. Declaring two invariants is equivalent to declaring a single invariant whose predicate conjoins the predicates declared in the two clauses. The first invariant clause says that the value of the `age` field is always between 0 and 150 (inclusive). Although this invariant is true for objects whose dynamic type is exactly `Animal`, it is not necessarily true for subtypes of `Animal`; a subtype could declare a method that would allow values outside this range to be assigned to `age`. Thus it is necessary to explicitly declare this invariant [55]. In effect, this invariant prohibits methods that set `age` outside the range specified in the invariant.

The second invariant clause in part documents a design decision, since it says that all elements of the List `history` are instances of type `String`. So it is closely related to what some authors call a "representation invariant" [32, 54]. However, since `history` is public for specification purposes, the invariant is public and visible to clients.

```
import java.util.*;
public class Patient extends Person {
  //@ public invariant 0 <= age && age <= 150;

  protected /*@ spec_public rep @*/ List history;
  /*@ public initially history.size() == 0;
    @ public invariant (\forall int i; 0 <= i && i < history.size();
    @                           history.get(i) instanceof rep String);
    @ public constraint \old(history.size()) <= history.size();
    @ public constraint (\forall int i; 0 <= i && i < \old(history.size());
    @                           history.get(i).equals(\old(history.get(i))));
    @*/

  /*@ requires !obs.equals("");
    @ assignable history.theCollection;
    @ ensures history.size() == \old(history.size()+1)
    @      && history.get(\old(history.size()+1)).equals(obs);   @*/
  public void recordVisit(String obs) {
    history.add(new /*@ rep @*/ String(obs));
  }

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender, history;
  //@ ensures gender.equals(g);
  public Patient(String g) { super(g); history = new /*@ rep @*/ ArrayList(); }
}
```

Fig. 5. A JML specification of the class `Patient`. The **`invariant`** clause in a class declares an invariant, and **`constraint`** declares a history constraint. The **`rep`** annotations declare ownership properties. JML's specification of `List` includes a data group named `theCollection`.

JML distinguishes instance invariants from static invariants. Instance invariants can refer to the state of an instance of the enclosing type using the keyword **`this`** and the names of instance (non-static) fields. Static invariants cannot refer to the state of an instance. Both of the invariants in Fig. 5 are instance invariants.

2.4.2 History constraints History constraints are taken from Liskov and Wing's work [55, 56], and specify a very simple kind of temporal property. They are used to declare monotonic relationships that are preserved by methods of a type.

The two **`constraint`** clauses in Fig. 5 declare two history constraints for the type `Patient`. (Again, having two history constraints is equivalent to having one constraint which conjoins the two predicates.) The first constraint says that the size of the `history` list never shrinks; that is, the size of `history` is monotonically non-decreasing. The second says that elements in the `history` list are never deleted.

In JML history constraints can be used to collect common postconditions, in much the same way that invariants can be used to collect common pre- and postconditions. For example, the ensures clause of the `recordVisit` method does not need to specify that the elements of `history` are preserved, as this is implicit in the second history constraint. This helps make specifications more understandable.

2.4.3 Initially clauses The **`initially`** clause in Fig. 5 on the previous page gives a predicate that is to be true in the post state of each (non-helper) constructor. It can thus be thought of as conjoined to the postcondition for `Patient`'s constructor. In JML initially clauses can be used to collect postconditions from constructors. While initially clauses are not involved in reasoning about dynamic dispatch, they are useful for reasoning with invariants and history constraints. When used with the invariants declared in a type, they provide a basis for datatype induction. When used with history constraints the provide a basis for computing the set of reachable object states. When an object is created its state must satisfy each declared initially clause. When its state is mutated, the method doing the mutation must satisfy each history constraint. Thus using an initially clause and a history constraint one may restrict the set of reachable states for a type and its subtypes in a way that would otherwise not be expressible.

In JML a type may have several initially clauses. As with invariants and history constraints, writing multiple initially clauses in a type specification is equivalent to writing one initially clause with the conjunction of their predicates. In the following the phrase "*the initially predicate*" for a type refers to this conjunction.

2.5 Specification Inheritance

In JML specifications of subtypes inherit not only fields and methods from their supertypes, but also specifications. Thus, to fully understand the examples given so far, you need to understand how JML's specification inheritance works.

To explain specification inheritance it helps to fix a bit of notation for type specifications. For a type T, let *added_inv*^{T} be the invariant predicate declared in T's specification (i.e., without inheritance), let *added_hc*^{T} be the history constraint declared in T's specification, and let *added_init*^{T} be the initially predicate declared in T's specification. Let *supers*(T) be the set of all supertypes of T (including T) and let *methods*$(\mathcal{T})$ be the set of all instance method names declared in the specifications of the types in a set $\mathcal{T}$. (For simplicity, I assume that statically overloaded methods have been distinguished by adding to each method name a list of the method's argument types; thus each method name is associated unambiguously with a list of argument types. I also assume that there is no shadowing of fields and that all overriding methods use the same formal parameter names as the methods they override; these assumptions can also be made with no loss of generality by use of renamings.)

For methods, I use the notation *added_spec*^T_m $=$ (*added_pre*^T_m, *added_post*^T_m) for the pre/post specification declared in type T for method m. Such a specification is the join of the specification cases specified in type T for m. If there are no specification cases in type T for method m, this notation should still be defined, but one has to distinguish two cases. If m is declared in T with no specification and is not overriding any

methods in T, then $\mathit{added\_spec}_m^T = (\mathit{true}, \mathit{true})$. This corresponds to the JML default specification, which places no limits on callers or on the implementation. However, if m is not declared in T (and hence has no specification in T), then we want a method specification that will not affect the join of other method specifications. Hence in this case we define $\mathit{added\_spec}_m^T = (\mathit{false}, \mathit{true})$, which is the identity with respect to the join of method specifications. Appropriately, this least useful specification is also the join of the empty set of method specifications, $\bigsqcup \emptyset$.

As in Java, a JML specification for a type inherits instance field declarations from its proper supertypes, including the modifiers (such as **`spec_public`**) and data group declarations that are part of such field declarations. This inheritance applies to model (and ghost) fields, as well as Java fields. As noted earlier, inheritance of such declarations is important for making sense of predicates inherited from supertypes. Represents clauses, which specify how to retrieve the values of model fields are also inherited in JML. Overriding of (functional) represents clauses in subtypes presents semantic problems [52], and thus I will assume that the type checker prohibits it. Since represents clauses and fields are merely collected and not combined like method specifications or invariants, I omit them from the definition below.

With these conventions, the mechanism JML uses to inherit specifications can be explained by constructing an extended specification [23, 42].

Definition 2 (Extended specification). *Suppose T has supertypes $\mathit{supers}(T)$, which includes T itself. Then* the extended specification of T *is a specification such that:*

methods: *for all methods $m \in \mathit{methods}(\mathit{supers}(T))$, the extended specification of m is the join of all added specifications for m in T and all its proper supertypes:*

$$\mathit{ext\_spec}_m^T = \bigsqcup^T \{\mathit{added\_spec}_m^U \mid U \in \mathit{supers}(T)\},$$

invariant: *the extended invariant of T is the conjunction of all added invariants in T and its proper supertypes:*

$$\mathit{ext\_inv}^T = \bigwedge \{\mathit{added\_inv}^U \mid U \in \mathit{supers}(T)\},$$

history constraint: *the extended history constraint of T is the conjunction of all added history constraints in T and its proper supertypes:*

$$\mathit{ext\_hc}^T = \bigwedge \{\mathit{added\_hc}^U \mid U \in \mathit{supers}(T)\},$$

initially predicate: *the extended initially predicate of T is the conjunction of all added initially predicates in T and its proper supertypes:*

$$\mathit{ext\_init}^T = \bigwedge \{\mathit{added\_init}^U \mid U \in \mathit{supers}(T)\}.$$

2.6 Examples of Specification Inheritance

Specification inheritance for invariants, history constraints, and initially clauses is simple. It simply conjoins the appropriate predicates from a type and its supertypes. For

example, the type FemalePatient specified in Fig. 6 would inherit these clauses from Patient (see Fig. 5 on page 10). The history constraints and initially predicates are inherited without change. However, the invariant of FemalePatient is the conjunction of the invariant added in Fig. 6 and the invariant of Patient (which is the conjunction of the two invariants in Fig. 5).

```
public class FemalePatient extends Patient {
  //@ public invariant gender.equals("female");

  //@ assignable gender;
  public FemalePatient() { super("female"); }
}
```

Fig. 6. A JML specification of the class FemalePatient.

Specification inheritance for methods simply joins together all the method specifications from a type and its supertypes. For example, the extended specification of the isFemale method of Gendered from Fig. 1 on page 4 is just the specification (true, Q), where Q is the postcondition from that figure. This is the extended specification because isFemale is not specified in any supertypes of Gendered. This same specification for isFemale, (true, Q), is inherited unchanged by Animal, because Fig. 2 does not have any added specification cases for isFemale, so $\mathit{added\_spec}^{\texttt{Animal}}_{\texttt{isFemale}}$ is the identity specification $(\mathit{false}, \mathit{true})$. Thus the extended specification for isFemale is $\bigsqcup\{(\mathit{true}, Q), (\mathit{false}, \mathit{true})\}$, which equals (true, Q). Similarly, isFemale has the same extended specification in the classes Person and Patient.

A more interesting example is the setAge method. This method is specified for the type Animal in Fig. 2 on page 5, and in its subtype Person in Fig. 4 on page 9. The extended specification of setAge in type Person is thus the join of these two specifications. (This join is also inherited by the type Patient specified in Fig. 5 on page 10.) Using the definitions given above, one can compute a single specification case that is equivalent to this join. However, when reading a JML specification with multiple specification cases, it is not necessary to calculate the specification of their join. Instead, the reader of such a specification just has to remember that each specification case must be obeyed by a correct implementation. For this reason, the jmldoc tool shows the join using **also** instead of the more complex, calculated specification. For example, compare the specification in Fig. 7 on the following page to that in Fig. 3 on page 8.

With specification inheritance it is impossible to make a method's precondition strictly stronger than what is inherited. Consider the class Senior specified in Fig. 8 on page 15. At first glance, the setAge method in Fig. 8 seems to specify a method with a stronger precondition than setAge's extended precondition in Person, which is a <= 150. However, taking specification inheritance into account, the extended precondition of setAge in Senior is the disjunction of a <= 150 and the precondition in the added specification case, and hence is equivalent to a <= 150. Thus the argument to setAge can legally be 18, for example, and in this case the Senior's age will be set to 18.

```
/@   requires 0 <= a && a <= 150;            // from Animal
 @   assignable age;
 @   ensures age == a;
 @ also
 @   requires a < 0;
 @   assignable age;
 @   ensures age == \old(age);
 @ also
 @   requires \same;                         // from Person
 @   assignable age, ageDiscount;
 @   ensures 65 <= age ==> ageDiscount;   @*/
public void setAge(int a);
```

Fig. 7. The join of the 3 specification cases for `setAge` for the type `Person`, presented as a join of specification cases. In such contexts the precondition `\same` means the disjunction of the other (non-`\same`) preconditions.

Findler and Felleisen [28] note that it might be better for the specification language to point out this situation as a problem. Since it is not possible with specification inheritance to strengthen an inherited precondition, it would be reasonable to disallow what seem like attempts to strengthen a method's precondition if there is no good use for writing such a precondition. One reason for stating a stronger precondition would be to say that some extra effects happen in a subset of the cases in which the method may be called, as shown in Fig. 9 on the next page. However, such examples can be specified without changing the precondition, as shown in Fig. 4 on page 9. Another reason for writing a stronger precondition would be to redundantly specify some effect of the method, to bring it to the reader's attention. However, JML has a way to mark redundant specification cases explicitly, by putting them in the **`implies_that`** section or **`for_example`** sections of a method specification [40, 47]. So it may be sensible for JML to at least give a warning if such a non-redundant method specification strengthens the inherited precondition.

3 Supertype Abstraction

Subtyping causes a fundamental problem for reasoning about object-oriented programs. The problem is that since one generally does not know the dynamic (runtime) type of an object, the specification the object obeys is also unknown. Early discussions of this problem focused on reasoning about dynamically-dispatched method calls [3, 4, 38, 43, 53], but the problem also applies to invariants, history constraints, and initially clauses.

To explain the reasoning problems caused by dynamic dispatch, consider Fig. 10 on the next page. In that example, the `isFemale` method of the `Gendered` interface is called on each element of the List argument `s`. This works even if the List contains objects of different dynamic (runtime) types, thanks to dynamic dispatch.

The pre- and postconditions in this specification use universal quantifiers. In JML a universal quantifier has the form (`\forall` $T\ x$`;` $R(x)$`;` $B(x)$), which is true when

```
public class Senior extends Person {
  /*@ also
    @   requires 65 < a && a <= 150;
    @   assignable age;
    @   ensures age == a;
    @*/
  public void setAge(final int a) { super.setAge(a); }

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender, age;
  //@ ensures gender.equals(g) && age == 66;
  public Senior(final String g) { super(g); age = 66; ageDiscount = true; }
}
```

Fig. 8. A JML specification of the class `Senior`.

```
  /*@ also
    @   requires 65 <= age;
    @   assignable age, ageDiscount;
    @   ensures ageDiscount;   @*/
  public void setAge(final int a);
```

Fig. 9. Specifying an extra effect in `Person`'s `setAge` method when `65 <= age`.

```
  /*@ requires (\forall nullable Object e; s.contains(e);
    @                                    e instanceof Gendered);
    @ ensures (\forall Gendered e; \result.contains(e);
    @                      s.contains(e) && e.gender.equals("female"));  @*/
  public List females(List s) {
    List r = new ArrayList();
    Iterator elems = s.iterator();
    while (elems.hasNext()) {
      Gendered e = (Gendered)elems.next();
      if (e.isFemale()) { r.add(e); }
    }
    return r;
  }
```

Fig. 10. A method that extracts a list of females.

for all x of type T, if the range predicate $R(x)$ holds, then $B(x)$ holds. The modifier **`nullable`** in the precondition is used to allow `e` to range over null as well as other objects. By default declarations in JML do not allow null as a value, but null is a possible element of a List in Java. Thus the precondition says that all the elements of the argument `s` must be instances of the type `Gendered` (and in particular not null). The postcondition says that all elements of the result were in the argument `s` and are female.

To reason about the functional correctness of the `females` method in Fig. 10 one has to know how to reason about calls to methods with possibly unknown specifications. For example, `e.isFemale()` calls a method with a possibly unknown specification because the exact dynamic type of `e` is unknown; all that is known is that it implements the interface `Gendered`. The receiver `e` could represent a person, animal, or German noun.

The technique of *supertype abstraction* [42, 43] uses the specification of the static type of the receiver to reason about such calls. Thus, since `e`'s static type is `Gendered`, supertype abstraction tells us to reason about the call `e.isFemale()` using the specification given in Fig. 1 on page 4. This specification has no precondition, so we can conclude that the call returns true just when the gender of `e` is `"female"`. This allows us to conclude that `e` is only added to `r` if it is female, which helps establish the postcondition of the method `females`.

However, supertype abstraction and the problems it solves are not limited to reasoning about method calls. The same technique of using static type information solves problems in reasoning that uses invariants and history constraints [55] and also in reasoning that uses initially predicates. For example, if `p` is a variable that has static type `Patient` (see Fig. 5 on page 10), then using supertype abstraction, one could look at the invariant declared in type `Patient`, and conclude that `p.age <= 150`. Without supertype abstraction this conclusion could only be made if one knew the invariant of the dynamic type of `p`. Similarly, supertype abstraction works with history constraints. For example, it would allow one to conclude, after invoking a method with receiver `p` of static type `Patient`, that the size of `p.history` has not become smaller. Finally, supertype abstraction works with initially predicates. For example, it would allow one to prove the assertion in the following code fragment.

```
Patient p;
if (B) { p = new Patient("male"); } else { p = new FemalePatient(); }
//@ assert p.history.size() == 0;
```

Supertype abstraction was essentially invented by the first object-oriented programmers. It embodies the idea that objects of all subtypes of a type (including that type itself) can be treated uniformly.[5] These programmers reasoned (informally) that whenever they added a new proper subtype of an existing type to their program, unchanged code would continue to work correctly even when it operated on these new objects. For example, if they added a proper subtype of `Gendered` to the program, they would expect that the method `females` would still work correctly on objects of this new type.

[5] Conversely, reasoning using supertype abstraction embodies this treatment of each method name and type as standing for a common behavior.

Supertype abstraction is so ingrained in object-oriented thinking that it is hard to imagine alternative ways of reasoning about dynamic dispatch. Yet doing so helps illustrate the benefits (and limitations) of supertype abstraction.

An alternative to using supertype abstraction is to use the specification of each possible dynamic type of an expression's value. For example, suppose we know that in a call to the method `females`, the argument `s` only contains objects of type `Person` (which must be a subtype of `Gendered`). Then we could use the specification of `Person`'s method `isFemale` to reason about the call `e.isFemale()`. If `e` might have dynamic types `Person` and `GermanNoun`, then we would have to consider two cases in the proof, one for each of these specifications. In general, if `e` can have n different types, we would have to consider n cases. The advantage of using supertype abstraction is that we avoid this case analysis, since we only use a single specification, namely the one associated with `Gendered`. The disadvantage of supertype abstraction is that, since it does no case analysis, it cannot exploit special properties of these subtypes, such as `Person` or `GermanNoun`. Supertype abstraction thus trades specificity and reasoning power for uniformity and simplicity of reasoning.

However, there is a way to sidestep this disadvantage of supertype abstraction by moving the case analysis into the program's code, using downcasts and type tests. An example of this idea is given in Fig. 11, which takes an object of static type `Gendered` that must dynamically have type `GermanNoun` (see Fig. 12 on the next page) or `GreekNoun` (which is similar, but not shown). In Fig. 11 **`instanceof`** tests are used to do a case analysis, and within the different cases the code does downcasts. Due to these downcasts, one can again use supertype abstraction to reason about the variables `gern` and `grkn`. In particular one can reason about the call `gern.isMale()` using the specification of `isMale()` in the type `GermanNoun`. And one can use the invariant of `GermanNoun` to conclude that if `gern` is neither female nor male, then it must be neuter. This shows how case analysis (in code) and supertype abstraction (in reasoning) can be used together. Thus insisting on supertype abstraction is not as limiting as it might at first appear.

```
/*@ requires n instanceof GermanNoun || n instanceof GreekNoun;
  @ ensures \result <==> n.gender.equals("neuter");   @*/
public boolean isNeuter(final Gendered n) {
  if (n instanceof GermanNoun) {
    GermanNoun gern = (GermanNoun) n;
    return !(gern.isFemale() || gern.isMale());
  } else {
    GreekNoun grkn = (GreekNoun) n;
    return !(grkn.isFemale() || grkn.isMale());
  }
}
```

Fig. 11. A method that uses downcasts so that reasoning about calls can use both the special properties of the dynamic types `GermanNoun`, `GreekNoun`, and supertype abstraction.

```
public interface GermanNoun extends Gendered {
  /*@ public instance invariant gender.equals("female")
    @              || gender.equals("male") || gender.equals("neuter");  @*/

  //@ ensures \result <==> gender.equals("male");
  /*@ pure @*/ boolean isMale();
}
```

Fig. 12. A JML specification of a type `GermanNoun`. The type `GreekNoun` is similar.

Another advantage of supertype abstraction is that it permits reasoning with fewer assumptions. In particular, reasoning that uses supertype abstraction can be valid without assuming knowledge of all possible dynamic subtypes. In other words, supertype abstraction does not need knowledge of a whole program, and permits reasoning about programs that are open to the addition of new subtypes. For example, supertype abstraction allows reasoning about the correctness of the method `females` using the specifications of `Gendered`, without the need to know what dynamic subtypes are possible. Supertype abstraction allows reasoning about calls such as `e.isFemale()` even before subtypes of `Gendered`, such as `Person`, have been written. In this sense supertype abstraction is a modular reasoning technique.

4 Behavioral Subtyping

JML is designed to make supertype abstraction valid by making each type a behavioral subtype of each of its supertypes. To do this, it uses specification inheritance [23, 42, 45, 47, 75, 79] and methodological restrictions on invariants, etc. [24, 62, 63].

Much of the material below is adapted from my work with Dhara [23] and Naumann [42]. Interested readers should consult the latter [42] for details and proofs. I follow it in defining behavioral subtyping using the concept of refinement of method specifications, and in discussing the property needed from a methodology for invariants, etc.

4.1 Refinement of Method Specifications

Refinement is a binary relation on method specifications [42, 61]. Recall that $T \triangleright \mathit{spec}$ is a specification of a method that type checks with a receiver of static type T.

Definition 3 (refinement w.r.t. T', $\sqsupseteq^{T'}$). *Let $T' \triangleright \mathit{spec}'$ and $T \triangleright \mathit{spec}$ be specifications of an instance method m, such that T' is a subtype of T. Then spec'* refines *spec* with respect to *T', written $\mathit{spec}' \sqsupseteq^{T'} \mathit{spec}$, if and only if for all calls of m where the receiver's dynamic type is a subtype of T', every correct implementation of spec' satisfies spec.*

The refining specification, *spec*′ is stronger than *spec* in the sense that it restricts implementations more than does *spec*. Thus it may be that fewer implementations satisfy *spec*′ compared to those that satisfy *spec*. From a client's point of view, *spec*′ may be more useful, while from the implementor's point of view *spec*′ may be more difficult.

In the above definition, the condition on the receiver's dynamic type allows a specification for method m in a subtype to refine m's specification in one of its supertypes. (This condition would be dropped if one were considering refinement of Java static methods or constructors, which have no receiver.)

For an example of refinement, I will show that the first specification case of `setAge` in Fig. 2 on page 5 is refined by the specification given in Fig. 3 on page 8.[6] Showing this refinement means showing that if an implementation satisfies the specification in Fig. 3, then it satisfies the specification for `setAge` given in Fig. 2. This is true, for example, of the implementation given in Fig. 2, which satisfies both specifications, due to the conditional. However, if this conditional were omitted and the method's body always assigned to `age`, then the body would still be a correct implementation of the specification in Fig. 2. However, it would not correctly implement the specification in Fig. 3. For example, with the omitted conditional, a call such as `setAge(-1)` would assign to `age`, possibly violating the first conjunct of the ensures clause in Fig. 3. It follows that the specification in Fig. 2 is not a refinement of the specification in Fig. 3.

4.1.1 Proving refinements To show that the specification in Fig. 3 really is a refinement of the first specification case in Fig. 2, one must show that every implementation that satisfies this specification satisfies the specification given in Fig. 2.

A general way to do such a proof is to prove a relationship between the specifications in question. Java and JML's type checking implies that if $T' \triangleright (\mathit{pre}', \mathit{post}')$ is to refine $T \triangleright (\mathit{pre}, \mathit{post})$, then T' must be a subtype of T. Furthermore, in Java, both must have the same argument types. For simplicity, I will assume that the formal parameter names are the same. I will also use the notation $\text{Spec}(T') \vdash P$ to mean that P is provable using the semantics of Java and the specification of T'. (Also, the notation `&&` means logical conjunction as in JML.) With these conventions we have the following theorem [13, 42, 50, 64, 68].

Theorem 1. *Let $T' \triangleright (\mathit{pre}', \mathit{post}')$ and $T \triangleright (\mathit{pre}, \mathit{post})$ be specifications of an instance method m, where T' is a subtype of T. Then $(\mathit{pre}', \mathit{post}') \sqsupseteq^{T'} (\mathit{pre}, \mathit{post})$ if and only if the following two conditions hold:*

$$\mathit{Spec}(T') \;\vdash\; \mathit{pre}\ \texttt{\&\& (this instanceof}\ T'\texttt{)} \implies \mathit{pre}' \tag{3}$$

$$\mathit{Spec}(T') \;\vdash\; \texttt{\textbackslash old(}\mathit{pre}\ \texttt{\&\& (this instanceof}\ T'\texttt{))} \implies (\mathit{post}' \implies \mathit{post}). \tag{4}$$

∎

Condition (3) says that the refinement's precondition *pre′* cannot make more assumptions than *pre*, except perhaps about the receiver's type. Since subtypes inherit the fields of their supertypes, *pre* makes sense for all objects of type T'. Note that if both specifications are for the same type, T', then Java guarantees the receiver is an instance of T' (or a subtype), and so in this case (3) just says that *pre* implies *pre′*. Condition (4) says that whenever a call whose receiver has type T' satisfies *pre*, and the refinement's postcondition *post′* is true, then *post* must hold. It can also be simplified if the

[6] This comparison ignores the second specification case in Fig. 2.

receiver types are the same (T'), since in that case we can again ignore the conjunct (`this instanceof` T').

In the `setAge` example, we can prove (3), because `0 <= a && a <= 150` implies the disjunction of that condition and `a <= 150`. And we can prove (4) because whenever `0 <= a && a <= 150` holds in the pre-state, and the postcondition of Fig. 3 holds, it follows that `age == a`. We can ignore the assignable clauses in this proof, since they are identical and in such a case JML's semantics implies that the translation of the assignable clauses will be the same.

An important point is that simplifying (4) by omitting its dependence on *pre* makes the notion of refinement too restrictive (i.e., unable to prove some refinements that meet the definition). For example, note that the postcondition of `setAge` in Fig. 3 does not imply the postcondition in the first specification case of Fig. 2. To see this, consider what happens if `a` is `-1`, in which case in the postcondition in Fig. 3 simplifies to `age == \old(age)`, which does not imply the postcondition in the first specification case of Fig. 2, `age == a`. However, as we have just shown, (4) does hold for this example. Thus a refining specification is unconstrained for states that do not satisfy the precondition of the specification it refines.

4.1.2 Refinement and assignable clauses Although assignable clauses can be considered as shorthand for part of a postcondition, it is useful to be able to treat them separately in a proof of refinement. To do this, suppose that the assignable clause of *spec*′ has the list L' and that the assignable clause of *spec* is L. Then one has to prove:

$$\begin{aligned} \mathrm{Spec}(T') \;\vdash\; & \texttt{\textbackslash old(}\mathit{pre}\ \texttt{\&\& (this instanceof}\ T'\texttt{))} \\ & \implies (\texttt{\textbackslash only\_assigned}(L') \implies \texttt{\textbackslash only\_assigned}(L)). \end{aligned} \tag{5}$$

Doing this allows one to omit the translation of the assignable clauses in the proof of (4). Informally, (5) means that the frame of *spec*′ can be more restrictive than that of *spec*, but data group membership has to be decided based on the specification of the refinement's receiver type, T'. That data group membership matters can be seen by considering Fig. 13, where the subtype `Animal`'s specification is needed to show that `gen` is a member of `gender`'s data group, and hence when at most the locations in the data groups of `gender` and `gen` are assigned, then at most the locations in `gender`'s data group are assigned [49].

4.1.3 Refinement of binary methods such as `equals` "Binary" methods, which operate on one or more arguments of the same type as the receiver [7], pose special pitfalls for refinement (and hence for behavioral subtyping [55]).

These pitfalls can be demonstrated by considering Java's `equals(Object)` method. For example, consider a specification for `Gendered`'s `equals` method as in Fig. 14. This is almost certainly an overly strong specification, since it allows no variation in refinements (and hence in subtypes). The specification says that when two objects that are subtypes of `Gendered` are compared, the method must return true just when their genders are equal, and it must return false otherwise. Thus, this specification says that the only attribute of an object of any subtype of `Gendered` that matters for `equals` is

```
//@ refines "Animal.refines-jml";
public class Animal implements Gendered {
  /*@ also
    @   protected behavior
    @     assignable gender, gen;
    @     ensures gen == g.equals("female");  @*/
  public Animal(String g);
}
```

Fig. 13. A JML refinement file that refines the specification of the constructor in Fig. 2 on page 5. The **refines** directive says that this file is to be used to refine the file `Animal.java`. The annotation **protected behavior** says that this is a specification of protected visibility.

```
  /*@ also
    @   ensures obj instanceof Gendered
    @       ==> \result == gender.equals(((Gendered)obj).gender);   @*/
  public /*@ pure @*/ boolean equals(/*@ nullable @*/ Object obj);
```

Fig. 14. A bad (unrefinable) specification of the `equals` method of type `Gendered`.

the object's gender. However, as in real life, other attributes do matter. For example, we might wish to distinguish two objects of type `Animal` if they have different ages or if they have different identities (i.e., if they are not ==). But, as the reader can check, such specifications are not refinements of the one in Fig. 14.

A better way to specify the `equals(Object)` method is shown in Fig. 15. This is a looser specification, since it allows the method to always return false. This freedom allows refinements (and hence subtypes) to specialize the method by considering other attributes of `Gendered` objects, such as their age or object identity. The specification in Fig. 15 says (in the first ensures clause) that for the case where the argument `obj` is an instance of `Gendered`, when the method returns true, then the argument must have the same gender as the receiver. The reader should check that this allows the method to return false even if the argument is an instance of `Gendered` and the genders are equal.

Two equivalent ways of writing this specification are given in the **implies_that** section of Fig. 15 [40, 47]. The first ensures clauses following **implies_that** says that when the argument is a `Gendered` object with a different gender, then the method returns false. The last redundant ensures clause is a logically equivalent way of writing the non-redundant ensures clause that follows **also**.

This problem of overspecifying the `equals` method mainly affects types with immutable objects, because for a type with mutable objects, the `equals` method should usually be specified to compare object identities. However, this problem does occur in real examples. For instance when we first specified the type `java.util.Date` we used a specification of its `equals` method that only allowed comparison of the millisecond times (written in a way similar to Fig. 14). However, this was too strong because there

```
/*@ also
  @   ensures obj instanceof Gendered
  @       ==> (\result ==> gender.equals(((Gendered)obj).gender));
  @ implies_that
  @   ensures obj instanceof Gendered
  @       ==> (!gender.equals(((Gendered)obj).gender) ==> !\result);
  @   ensures obj instanceof Gendered && \result
  @       ==> gender.equals(((Gendered)obj).gender);      @*/
public /*@ pure @*/ boolean equals(/*@ nullable @*/ Object obj);
```

Fig. 15. A good (refinable) specification of the `equals` method for the type `Gendered`. The section following **`implies_that`** states redundant consequences of the specification. This **`implies_that`** section can be omitted without changing the meaning of the specification.

could be subtypes, that need to distinguish objects based on other attributes, such as a number of nanoseconds.

4.1.4 Using `also` to make refinements JML makes sure that an implementation refines all specification cases given for it by joining them together. This is the reason for the using **`also`** in the syntax to connect specification cases. The connection between the join of specification cases using **`also`** and refinement is shown in the following nice little theorem [13, 23, 40, 42, 50, 64, 80]. The proof assumes that **`\old()`** is *monotonic* in the sense that: $(Q \implies P) \implies (\texttt{\textbf{\textbackslash old}}(Q) \implies \texttt{\textbf{\textbackslash old}}(P))$.

Theorem 2. *Suppose* **`\old()`** *is monotonic. Let* $T' \triangleright (pre', post')$ *and* $T \triangleright (pre, post)$ *be specifications of an instance method* m*, where* T' *is a subtype of* T*. Then*

$$((pre', post') \sqcup^{T'} (pre, post)) \sqsupseteq^{T'} (pre, post).$$

Proof: Let m, $T' \triangleright (pre', post')$, and $T \triangleright (pre, post)$, be as stated. Theorem 1 gives two conditions to prove using Spec(T'). To show (3) we can calculate as follows.

$$\begin{array}{ll} & pre \texttt{ \&\& (this instanceof } T'\texttt{)} \\ \implies & \langle \text{by } (P \texttt{ \&\& } I) \implies P \rangle \\ & pre \\ \implies & \langle \text{by } P \implies (P' \texttt{ || } P) \rangle \\ & (pre' \texttt{ || } pre) \end{array}$$

To show (4) assume that **`\old`**(*pre* **`&& this instanceof`** T'`)` holds. Since **`\old()`** is monotonic by assumption, **`\old`**(*pre*`)` holds. Now we can calculate as follows.

$$\begin{array}{ll} & (\texttt{\textbf{\textbackslash old}}(pre')\texttt{ ==> } post') \texttt{ \&\& } (\texttt{\textbf{\textbackslash old}}(pre)\texttt{ ==> } post) \\ \implies & \langle \text{by } (X' \texttt{ \&\& } X) \implies X \rangle \\ & \texttt{\textbf{\textbackslash old}}(pre)\texttt{ ==> } post \\ \implies & \langle \text{by assumption that } \texttt{\textbf{\textbackslash old}}(pre) \text{ holds} \rangle \end{array}$$

post ∎

A more involved proof is needed to show that there is no better definition of the join of method specifications; i.e., that the join of method specifications is their least upper bound in the refinement ordering [13, 42, 50, 64]. This justifies the notation "⊔".

4.1.5 Methodologies for invariants Besides refinement of method specifications, behavioral subtyping involves the other elements of a type specification. Initially clauses and history constraints have not been studied in much detail in academic papers, but they are similar enough to invariants that most research on invariants should apply to them. By contrast, invariants have been the subject of much recent research in object-oriented programming methodology [5, 51, 62, 63, 65, 66]. The reason that invariants are such a focus of research is that they have interesting interactions with aliasing, reentrance, and subtyping.

Aliasing can cause problems if objects contained in an object o are exposed to clients, who may break o's invariant without calling one of o's methods.

Reentrance causes problems for invariants when a method being run on some receiver object o breaks an invariant temporarily, and then while still running, makes a call that (eventually) runs a method whose receiver is o. In such a situation, the invariant may not hold in the pre-state of the call back to o.

Subtyping causes problems because, in a subtype, invariants can be strengthened [55]. However, since they can also be thought of as conjoined to the preconditions (and postconditions) of instance methods, this means that a stronger invariant in a subtype will strengthen the subtype's precondition. But, as described in condition (3) of Theorem 1 on page 19, strengthening the precondition of a refining specification is not allowed. To see the problem, consider the dynamic dispatch code in Fig. 10 on page 15. When the call is made to `e.isFemale()` and `e` has dynamic type `Patient`, how do we know that the invariant of `Patient` holds in the pre-state of the execution of `isFemale`?

To resolve these problems, the essential insight is that some set of restrictions on programs, i.e., a programming methodology, is needed. A programming methodology must validate the implicit assumption that each invariant holds in each (non-helper) method's pre-state [42, §2.3]. A programming methodology that allows one to safely assume invariants in pre-states is needed to validate reasoning with supertype abstraction, even if invariants cannot be weakened in behavioral subtypes [42, Lemma 23].

There are, broadly speaking two general approaches that are being investigated for such programming methodologies in the context of JML-like specification languages.

The first is the relevant invariant semantics [62, 63], which is based on an ownership type system [24]. Ownership is used both to prevent problems of representation exposure [54, 62, 66] and to deal with layered abstractions. Reentrance is dealt with by mandating that invariants are established at the point of calling a (non-helper) method.

This approach is being investigated in the context of JML. Dietl and Müller have integrated the Universe type system [24] into the JML checker, which can use ownership annotations to check that specifications follow the methodology. In particular the checker uses the **`rep`** annotations to indicate when contained objects are owned by an enclosing object. For example, in Fig. 5 the **`rep`** annotation in the declaration of `history` says that `history` is owned by the enclosing `Patient` object; i.e., that `history` is part

of the representation of `Patient`. The construction of a rep `String` object in the body of `recordVisit` places the newly created string in the `Patient` object's universe (ownership domain). Similarly, the **`rep`** annotation in the constructor's **`new`** expression says that the new object is in its owner's universe. Type checking ensures that invariants only depend on the state of owned objects and are never violated outside of the classes in which they are declared. For example, the type system checks that the object referred to by `history` is never exposed directly to clients, which would allow them to mutate it in ways that would violate the invariant or history constraint of `Patient`.

The second approach is the "Boogie" methodology [5, 51, 65], which is used in the specification language Spec#. To explain the Boogie methodology briefly, I will translate it into JML terms. Suppose each object has a ghost field, which I will call `validFor`. This ghost field could be declared in JML's specification of `Object` as follows.

```
//@ public ghost Class validFor = null;
```

In JML a ghost field is a specification-only field, like a model field, but which is not an abstraction of concrete fields. Instead, a ghost field is manipulated by using **`set`** statements, which are written in annotations and thus considered part of the program's specification. However, the `validFor` field is special in that the Boogie methodology only allows it to be assigned by two special statements **pack**(T) and **unpack**(T).

This field is used to weaken each declared invariant as follows. Suppose a type T declares an instance invariant inv^T. The Boogie methodology transforms this invariant into an implication: **`this`**`.validFor <: \type(`T`) ==>` inv^T. (In JML the operator **`<:`** means "is a subtype of" and **`\type()`** is used to enclose type names in expressions.) Thus this transformed invariant says that the declared invariant, inv^T, only has to hold when **`this`**`.validFor` is a subtype of T. In the Boogie methodology, this transformed invariant holds in every state, including the pre-state of each method. This is fundamental to solving the invariant problems.

In the Boogie methodology, one can only assign to the fields of an object o that are declared in a type T when an object is "unpacked for T," meaning that o`.validFor` is not a subtype of T. Unpacking an object is the job of the **unpack**(T) statement. When done changing an object's fields, one uses the **pack**(T) statement to check inv^T and to set `validFor` to T. Thus, whenever the program is able to assign to the fields of an object, that object must be unpacked, and hence the declared invariant does not have to hold. This may seem complicated, but the special statements are often implicitly wrapped around the body of a method using default annotations in Spec#.

Because it is based on dynamic manipulations of the `validFor` field, the Boogie methodology is more flexible than the relevant invariant approach. For example, the declared invariants do not have to be re-established on each call to a (non-helper) method, since the object's `validFor` field can be used to dynamically test whether the declared invariant holds. However, as one can see from this translation, some of these ideas (like the dependency of an invariant of part of a program's state) can be used in JML to gain some of the flexibility of the Boogie methodology. Whether these approaches can be usefully blended together is an interesting problem for future research.

Fortunately, the validity of supertype abstraction does not depend on the details of these methodologies. All that is needed is that they allow one to safely assume invariants in the pre-states of non-helper methods [42].

4.1.6 Semantic implication for objects of a type Predicates used in invariants, history constraints, and initially clauses written in the specification of a type T are written to use the fields (including model fields) and instance methods of that type. Because these are inherited by all subtypes of T, they make sense for all subtypes of T. In the following we will say that a predicate P is for T to describe this association between a predicate and this type context; technically P *is for* T if P type checks in the context of T, assuming that **this** has static type T. Note that if P is for T and T is a supertype of T', then P is also for T'. This notion is used in comparing the relative strength of invariants and history constraints.

Definition 4 (Implies for objects of type T'). *Let P' and P be predicates that are for a type T'. Then P'* implies *P* for objects of type *T' if and only if whenever* **this** *has a dynamic type that is a subtype of T' and P' holds, then P holds.*

It is a corollary that P' implies P for objects of type T' if and only if:

$$\text{Spec}(T') \vdash \texttt{this instanceof}\, T' \implies (P' \implies P). \tag{6}$$

4.2 A Definition of Behavioral Subtyping for JML

The following definition of behavioral subtyping strays a bit beyond the technical results in Leavens and Naumann's recent work [42] because the definition also treats history constraints and initially clauses. They only prove define behavioral subtyping for types with pre/post method specifications and invariants. However, in adapting their definition to JML I have followed the ideas in their work, which should again be consulted for details.

JML supports two notions of behavioral subtyping. There is an experimental notion of "weak behavioral subtyping" [20, 22, 23]. However, that notion relies on an untested programming methodology [21] which JML does not currently enforce. Thus the most important notion of behavioral subtyping for JML, which corresponds to Liskov and Wing's constraint-based definition [55, p. 1823], is the following.

Definition 5 (strong behavioral subtype). *Let T' be a type specification and let T be a specification for a supertype of T'. Then T'* is a strong behavioral subtype of *T if and only if:*

methods: *for all instance methods m in T, the method specification for m in T' refines that of m in T with respect to T',*

invariant: *the instance invariant of T' implies the instance invariant of T for objects of type T',*

history constraint: *the instance history constraint of T' implies the instance history constraint of T for objects of type T', and*

initially predicate: *the initially predicate of T' implies the initially predicate of T for objects of type T'.*

Notice that the definition above says nothing directly about constructors and thus applies equally well to Java interfaces. However, as Liskov and Wing emphasized [55, 56],

constructors are constrained by the invariant of each type. Furthermore, the **initially** predicate in a type specification also constrains constructors.

Normally the concept defined above will be referred to as "behavioral subtyping." However, it is useful to keep in mind that the above definition is designed for JML and how one reasons about JML programs using supertype abstraction. As discussed in the next subsection, when working with a specification language X, one needs a definition of behavioral subtyping that validates X's notion of supertype abstraction [2, 38]. Thus there really is no single, normative definition of behavioral subtyping.

4.3 Connection to Supertype Abstraction

The fundamental property of a definition of behavioral subtyping is that it makes supertype abstraction valid [38, 42, 43]. Ideally, a definition would also be no stronger than needed to make supertype abstraction valid. For example, since calls to constructors and static methods are not directly involved in reasoning using supertype abstraction, there is no need for a syntactic (or type) relationship between the constructors and static methods of a behavioral subtype and its supertypes. However, the definition must indirectly limit constructors and static methods (e.g., by enforcing invariants) so that supertype abstraction is valid.

Thus, ideally, behavioral subtyping would be both necessary and sufficient for supertype abstraction to be valid. To prove a theorem about this requires a precise formulation of supertype abstraction. Leavens and Naumann [42] have given such a precise formulation for reasoning with pre/post specifications about dynamically dispatched calls (i.e., in the absence of invariants, history constraints, and initially predicates). Their formulation uses two semantics for such calls, the normal (dynamic) one and a static one. With this notion of supertype abstraction, they shown that it is both necessary and sufficient that each (non-abstract) class be a behavioral subtype of all its supertypes [42, Corollary 13]. Somewhat surprisingly, it turns out that it is not necessary to have an interface (or an abstract class) be a behavioral subtype of its supertypes. They conjecture that with a suitable definition of supertype abstraction (i.e., one that allows reasoning about invariants based on static type information) there is again such an equivalence for specifications with invariants. However, their formal treatment only gives soundness in this case, using some invariant methodology that validates the assumption of invariants in method pre-states (see Section 4.1.5 on page 23).

The notion of supertype abstraction for JML described in this paper involves reasoning using pre/post specifications, invariants, history constraints, and initially predicates. The definition of behavioral subtyping is designed to make the following true.

Conjecture 1 (Supertype abstraction valid). Suppose JML enforces sensible methodological restrictions on invariants, history constraints, and initially predicates.

Then supertype abstraction for JML is valid if and only if each non-abstract class C is a behavioral subtype of all of its supertypes.

Proving a technically precise version of this conjecture would be an important check on the definitions of the programming methodology, supertype abstraction, and strong behavioral subtyping.

Although it is not necessary for the soundness of supertype abstraction, most treatments of behavioral subtyping make interfaces and abstract classes also be behavioral subtypes of their supertypes. JML does this also, through specification inheritance.

4.4 Connection to Specification Inheritance

With specification inheritance each subtype is forced to be a behavioral subtype of each of its supertypes [23, 42]. The following uses the notation from Section 2.5 on page 11.

Theorem 3 (Specification inheritance forces behavioral subtyping). *Let T and V be types where T is a subtype of V. Then the extended specification of T is a strong behavioral subtype of the extended specification of V.*

Proof: Let T and $V \in \mathit{supers}(T)$ be given. We must show that the extended specifications of T and V satisfy Definition 5 on page 25.

methods: Let m be an instance method in $\mathit{methods}(V)$. We show that $\mathit{ext\_spec}_m^T$ refines $\mathit{ext\_spec}_m^V$ with respect to T by the following calculation.

$$
\begin{array}{ll}
 & \mathit{ext\_spec}_m^T \\
= & \quad \langle \text{by Definition 2} \rangle \\
 & \bigsqcup^T \{\mathit{added\_spec}_m^U \mid U \in \mathit{supers}(T)\} \\
= & \quad \langle \text{by set theory, to separate out } V \text{ and its supertypes} \rangle \\
 & \bigsqcup^T \{\mathit{added\_spec}_m^U \mid U \in ((\mathit{supers}(T) \setminus \mathit{supers}(V)) \cup \mathit{supers}(V))\} \\
= & \quad \langle \text{by definition of join with respect to } T \rangle \\
 & \quad \left(\bigsqcup^T \{\mathit{added\_spec}_m^U \mid U \in (\mathit{supers}(T) \setminus \mathit{supers}(V))\}\right) \\
 & \sqcup^T \left(\bigsqcup^V \{\mathit{added\_spec}_m^W \mid W \in \mathit{supers}(V)\}\right) \\
\sqsupseteq^T & \quad \langle \text{by Theorem 2 on page 22, since } T \text{ is a subtype of } V \rangle \\
 & \bigsqcup^V \{\mathit{added\_spec}_m^W \mid W \in \mathit{supers}(V)\} \\
= & \quad \langle \text{by Definition 2} \rangle \\
 & \mathit{ext\_spec}_m^V
\end{array}
$$

invariant: We calculate as follows.

$$
\begin{array}{ll}
 & \mathit{ext\_inv}^T \\
= & \quad \langle \text{by Definition 2} \rangle \\
 & \bigwedge \{\mathit{added\_inv}^U \mid U \in \mathit{supers}(T)\} \\
= & \quad \langle \text{by set theory, to separate out } V \text{ and its supertypes} \rangle \\
 & \bigwedge \{\mathit{added\_inv}^U \mid U \in ((\mathit{supers}(T) \setminus \mathit{supers}(V)) \cup \mathit{supers}(V))\} \\
= & \quad \langle \text{by definition of conjunction} \rangle \\
 & \quad \left(\bigwedge \{\mathit{added\_inv}^U \mid U \in (\mathit{supers}(T) \setminus \mathit{supers}(V))\}\right) \\
 & \wedge \left(\bigwedge \{\mathit{added\_inv}^W \mid W \in \mathit{supers}(V)\}\right) \\
\Rightarrow & \quad \langle \text{by } A \wedge B \implies B \rangle \\
 & \bigwedge \{\mathit{added\_inv}^W \mid W \in \mathit{supers}(V)\} \\
= & \quad \langle \text{by Definition 2} \rangle \\
 & \mathit{ext\_inv}^V
\end{array}
$$

history constraint and initially predicate: these implications follow by the same reasoning as the implication for the invariant above. ∎

4.5 Examples of Behavioral Subtyping

The above theorem shows that, with specification inheritance, subtypes may only refine and strengthen specifications they inherit from their supertypes. However, specification inheritance can easily cause subtypes to not be satisfiable. For example, the invariant of class `OldAnimal` specified in Fig. 16 can be violated by the inherited `setAge` method, which is unsatisfiable, since no implementation of `setAge` will be able to both satisfy the inherited specification case and the added invariant.

```
public class OldAnimal extends Animal {
  //@ public invariant 65 < age;

  //@ requires g.equals("female") || g.equals("male");
  //@ assignable gender, age;
  //@ ensures gender.equals(g) && age == 66;
  public OldAnimal(String g) { super(g); age = 66; }
}
```

Fig. 16. A JML specification of the class `OldAnimal`.

However, it is possible to strengthen an invariant without making the specification unsatisfiable, as shown in the type `FemalePatient` from Fig. 6 on page 13. Liskov and Wing would call this type a "constrained" behavioral subtype [55] of `Patient` (see Fig. 5 on page 10). `FemalePatient`'s invariant limits the values of the model field `gender` to be the string `"female"`. Unlike the situation with the strengthened invariant in the type `OldAnimal`, there are no inherited methods that can change the `gender`, and hence this added invariant does not make the extended specification unsatisfiable.

In addition to constraining choices allowed by supertypes, a behavioral subtype may also add information and methods. Such a type is an "extension subtype" in Liskov and Wing's terminology [55]. The class `Dog`, given in Fig. 17 on the next page, extends the type `Animal` in this sense. `Dog`'s added invariant allows the specification of the method `setAge` to be inherited without change. This invariant implies that in its supertype, since by specification inheritance it is the conjunction of the added invariant and `Animal`'s invariant, which is just the default (true). This subtype also adds method `getDogAge`.

5 Related Work

The present paper is based on a recent semantical account that has a formal treatment of supertype abstraction and proves results about its connection to behavioral subtyping and specification inheritance [42]. The following draws on that paper's more detailed discussion of related work.

Several program logics for sequential Java incorporate a notion of supertype abstraction [62, 69, 70, 72]. They mostly require each overriding method implementation

```
public abstract class Dog extends Animal {
  public static final int D2PY = 7;   // conversion factor
  private /*@ spec_public @*/ int dogAge = 0; //@ in age;
  //@ public invariant dogAge == D2PY*age;

  //@ assignable \nothing;
  //@ ensures \result == dogAge;
  public int getDogAge() { return dogAge; }

  public void setAge(final int a) { super.setAge(a); dogAge = D2PY*age; }
  /* ... */
}
```

Fig. 17. A JML specification of the class Dog.

in a type to satisfy the corresponding specification in each of its supertypes, which is effectively the same as specification inheritance.

Liskov and Wing's paper [55] also discusses the idea of supertype abstraction to some extent. Their "subtype requirement" [55, p. 1812], says that properties of a supertype hold for all subtypes. However, the properties they consider are only those obtainable by inductive reasoning with invariants and history constraints, because they consider concurrent programs and do not require alias control. Due to concurrency their subtype requirement does not encompass the use of supertype abstraction to do pre/post reasoning about the correctness of method implementations, although their definition of behavioral subtyping is adequate for such reasoning if one were to consider a sequential language and impose a methodology to deal with the problems of invariants described in Section 4.1.5 on page 23. Liskov and Wing's formalization of behavioral subtyping uses abstraction functions. Abstraction functions are not needed in the formalization presented here, because all fields (including model fields) are inherited in JML, which makes the predicates used to specify supertypes automatically meaningful in subtypes. They give many interesting examples of their notion of behavioral subtyping.

Dhara and Leavens [23] explained specification inheritance for Larch/C++ and gave the first proof that it forces behavioral subtyping.

Wills introduced the idea of specification inheritance for combining "capsules" in his Fresco system [79]. In Fresco one can write several "capsules" for a method, which must all be obeyed by a correct implementation. Specification cases in JML are based on this idea. The idea of combining separate specification cases first appeared in Wing's dissertation [80]. That work introduced the Larch family of behavioral interface specification languages [30, 81], which were a precursor of JML.

Eiffel [59], another precursor of JML, also has behavioral subtyping and a form of specification inheritance. Mitchell and McKim describe an idea similar to the join of method specifications in their chapter on inheritance [60, Chapter 6].

Early work on behavioral subtyping is surveyed in a paper by Leavens and Dhara [41], including the work of America [3, 4], which has the first proof of the soundness of reasoning in the context of behavioral subtyping.

6 Conclusions

JML is a cooperative effort to enhance the utility of specification languages and associated tools. While the concepts presented in this paper seem well established, many challenges remain [46]. The main future work related to the present paper is limiting the notion of specification inheritance by warning where it appears that the specifier is trying to strengthen the precondition of an overriding method's specification [28]. Static analysis tools for JML could also warn when a subtype's specification was inconsistent, due to conflicts between inherited and added specifications. More work on JML's semantics, including a proof of Conjecture 1 on page 26 would also be interesting.

Specification inheritance in JML forces all subtypes to be behavioral subtypes. This ensures that one can use supertype abstraction to do modular reasoning using static type information. The key feature of JML that supports specification inheritance is JML's **`also`**, which automatically produces a refinement of the specification cases that it joins.

These ideas can also be used informally [54]. For example, when writing informal documentation for a method, one can mimic JML's use of **`also`** by starting with a phrase like "In addition to the inherited behavior, this method . . . "

Similarly, when designing a type as a subtype of various classes and interfaces, one can keep in mind the demands of behavioral subtyping [19, 37, 59]. For example one has to be careful not to strengthen the invariant of a class in a way that would contradict the specification of inherited methods. One should be especially careful not to overspecify when specifying binary methods, such as the `equals` method, which would make behavioral subtypes unable to consider additional attributes.

Finally, the notion of behavioral subtyping validates informal reasoning based on static type information. When the specifications associated with static types are not sufficient to draw a desired conclusion, one can use type tests and downcasts to record the need for stronger assumptions about the types of objects. This blends special case reasoning with the uniformity of supertype abstraction.

7 Acknowledgments

Special thanks to David Naumann, who co-developed the theory behind this paper with me [42], and with whom I have had many interesting conversations about this paper's topics. Thanks also to my other collaborators on topics related to behavioral subtyping: William Weihl, Krishna Kishore Dhara, Cesare Tinelli, and Don Pigozzi. Thanks to Barbara Liskov for starting me on the topic of object-oriented programming, and for her inspirational examples of how to explain ideas for programmers. Thanks to Jeannette Wing for her work on Larch, and for suggesting the Larch/C++ project, which eventually led to JML. Thanks to Yoonsik Cheon for joint work on Larch/C++ and to Al Baker, Clyde Ruby, and Tim Wahls for their collaboration on the initial design of JML, including the core features described in this paper. Thanks to Patrice Chalin, Yoonsik Cheon, Curtis Clifton, David Cok, Joseph Kiniry, Rustan Leino, Peter Müller, Arnd Poetzsch-Heffter, Erik Poll and the rest of the JML community (`jmlspecs.org`) for many discussions about JML, its design, semantics, and tool support. Thanks to Samik Basu, Kristina Boysen, David Cok, Faraz Hussain, David Naumann, Hridesh Rajan, Clyde Ruby, and Tim Wahls for comments on earlier drafts of this paper.

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] S. Alagic and S. Kouznetsova. Behavioral compatibility of self-typed theories. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 585–608, Berlin, June 2002. Springer-Verlag.

[3] P. America. Inheritance and subtyping in a parallel object-oriented language. In J. Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, volume 276.

[4] 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.

[5] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6): 27–56, 2004. URL `http://tinyurl.com/m2a8j`.

[6] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, Oct. 1995.

[7] K. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[8] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, Aug. 1999. http://www.abo.fi/~mbuechi/publications/TR297.html.

[9] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer, Sept. 2003.

[10] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[11] N. Cataño and M. Huisman. Formal specification of Gemplus's electronic purse case study. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002*, volume LNCS 2391, pages 272–289. Springer-Verlag, 2002.

[12] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, Lecture Notes in Computer Science. Springer-Verlag, 2006. URL `http://tinyurl.com/o4nxa`.

[13] 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.

[14] Y. Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, Apr. 2003. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR03-09/TR.pdf`. The author's Ph.D. dissertation.

[15] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[16] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H. R. Arabnia and Y. Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf`.

[17] Y. Cheon and G. T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.

[18] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, May 2005.

[19] W. R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. *ACM SIGPLAN Notices*, 27(10):1–15, Oct. 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[20] K. K. Dhara. Behavioral subtyping in object-oriented languages. Technical Report TR97-09, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames IA 50011-1040, May 1997. The author's Ph.D. dissertation.

[21] K. K. Dhara and G. T. Leavens. Preventing cross-type aliasing for more practical reasoning. Technical Report 01-02a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR01-02/TR.pdf`. Available from archives.cs.iastate.edu.

[22] K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. Available from `http://www.sciencedirect.com/science/journal/15710661`.

[23] 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 ISU CS TR #95-20c, `http://tinyurl.com/s2krg`.

[24] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, Oct. 2005. URL `http://www.jot.fm/issues/issue_2005_10/article1.pdf`.

[25] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

[26] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part II: Specifying components in RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):29–39, Oct 1994.

[27] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[28] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1–15, Oct. 2001.

[29] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 17–19 2002. ACM Press.

[30] 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.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, Oct. 1969.

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

[33] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Computing Science Institute, University of Nijmegen, 2003. URL `http://www.cs.kun.nl/research/reports/full/NIII-R0318.ps.gz`.

[34] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 329–340. ACM, Oct. 1998.

[35] C. B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[36] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2004.

[37] W. R. LaLonde, D. A. Thomas, and J. R. Pugh. An exemplar based Smalltalk. *ACM SIGPLAN Notices*, 21(11):322–330, Nov. 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[38] G. T. Leavens. Verifying object-oriented programs that use subtypes. Technical Report 439, Massachusetts Institute of Technology, Laboratory for Computer Science, Feb. 1989. The author's Ph.D. thesis.

[39] G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[40] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999. URL `http://tinyurl.com/qv84o`.

[41] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000. URL `http://www.cs.iastate.edu/~leavens/FoCBS-book/06-leavens-dhara.pdf`.

[42] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Aug. 2006. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR06-20/TR.pdf`.

[43] 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.

[44] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, Mar. 2005. URL `http://dx.doi.org/10.1016/j.scico.2004.05.015`.

[45] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006. `http://doi.acm.org/10.1145/1127878.1127884`.

[46] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. Technical Report 06-14, Department of Computer Science, Iowa State University, Ames, Iowa, May 2006. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR06-14/TR.pdf`.

[47] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, Jan. 2006.

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

[49] 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.

[50] K. R. M. Leino and R. Manohar. Joining specification statements. *Theoretical Comput. Sci.*, 216(1-2):375–394, Mar. 1999.

[51] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[52] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2006. URL `http://tinyurl.com/pzll8`.

[53] B. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[54] B. Liskov and J. Guttag. *Program Development in Java*. The MIT Press, Cambridge, Mass., 2001.

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

[56] 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).

[57] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1-2):89–106, Jan.–Mar. 2004.

[58] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[59] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[60] R. Mitchell and J. McKim. *Design by Contract by Example*. Addison-Wesley, Indianapolis, IN, 2002.

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

[62] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. URL `http://tinyurl.com/jtwot`.

[63] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, Oct. 2006. URL `http://dx.doi.org/10.1016/j.scico.2006.03.001`.

[64] D. A. Naumann. Calculating sharp adaptation rules. *Inf. Process. Lett.*, 77:201–208, 2001.

[65] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 2006. To appear.

[66] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[67] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct. 1994.

[68] E. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Comput. Sci.*, 24:337–347, 1983.

[69] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. URL `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-654.pdf`. The author's Ph.D. dissertation.

[70] C. Pierik. *Validation Techniques for Object-Oriented Proof Outlines*. PhD thesis, Universiteit Utrecht, 2006. URL `http://igitur-archive.library.uu.nl/dissertations/2006-0502-200341/index.htm`.

[71] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997. URL `http://tinyurl.com/g7xgm`.

[72] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium un Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999. URL `http://tinyurl.com/krjle`.

[73] A. Poetzsch-Heffter, P. Müller, and J. Schäfer. The Jive tool. `http://softech.informatik.uni-kl.de/twiki/bin/view/Homepage/Jive`, Apr. 2006. Checked August 2, 2006.

[74] E. Poll. A coalgebraic semantics of subtyping. In H. Reichel, editor, *Coalgebraic Methods in Computer Science (CMCS)*, number 33 in Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, 2000.

[75] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03e, Iowa State University, Department of Computer Science, May 2005. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR00-03/TR.pdf`.

[76] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 28 number 5 of *SIGSOFT Softw. Eng. Notes*, pages 267–276. ACM, 2003.

[77] Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420. Springer-Verlag, 2004. ISBN 3-540-21299-X.

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

[79] A. Wills. Specification in Fresco. In Stepney et al. [78], chapter 11, pages 127–135.

[80] J. M. Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[81] J. M. Wing. Writing Larch interface language specifications. *ACM Trans. Prog. Lang. Syst.*, 9(1):1–24, Jan. 1987.