Modular Enforcement of Supertype Abstraction and Information Hiding with Client-Side Checking

Henrique Rebêlo, Gary T. Leavens, and Ricardo Lima

CS-TR-12-03
January 2012

Keywords: Object-oriented programming; Runtime assertion checking; Modularity; Modular reasoning; Supertype abstraction; Information Hiding.

2011 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, testing tools, theory; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

This is a preprint of a paper that is submitted for publication.

Dept. of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

Modular Enforcement of Supertype Abstraction and Information Hiding with Client-Side Checking

Henrique Rebêlo, Gary T. Leavens, and Ricardo Lima

Federal University of Pernambuco, Brazil
{hemr,rmfl}@cin.ufpe.br
University of Central Florida, USA
leavens@eecs.ucf.edu

Abstract. Static reasoning tools for object-oriented (OO) languages use supertype abstraction, by verifying calls to methods using the specification associated with the receiver's static type. Unfortunately, contemporary runtime assertion checkers for OO are inconsistent with such static reasoning tools, since they check assertions in an overly-dynamic way on the supplier side. For method calls, such supplier-side checking occurs at the exact runtime type of the receiver object, which in general can be a proper subtype of the receiver object's static type. Since such a subtype can have a refinement of the corresponding supertype's specification, this specification difference can cause an inconsistency between runtime assertion checking and static verification tools. We explain how our technique of client-side checking allows runtime assertion checkers to use the specifications associated with static types, gaining consistency with static verification tools. Another advantage of such client-side checking is that it provides a way for runtime assertion checkers to use privacy information associated with specifications, which promotes information hiding.

1 Introduction

Runtime assertion checking (RAC) is a useful technique for finding errors in a program [8]. It is popular because it can be used to find where a program does not meet its functional specification at many stages of the development or maintenance process. Several behavioral interface specification languages, including Eiffel [24], the Java Modeling Language (JML) [17], Spec# [2], and Code Contracts [11], have RAC tools.

In object-oriented (OO) programming, RAC faces additional complications due to subtyping and dynamic dispatch. These features are problematic for reasoning about OO programs because subtypes may satisfy different specifications than their supertypes. Thus when dynamic dispatch selects a method to execute at the site of a polymorphic call, the set of method codes, to which the call may dispatch, do not necessarily satisfy a single specification. One way to reason about such polymorphic method calls would be to use an exhaustive case analysis, taking into account all possible dynamic types for the method's receiver object. However, that would be non-modular, as it would involve whole-program knowledge of all subtypes, and it would lead to re-verification whenever new subtypes are added to the program.

To tackle this modularity problem, Leavens and Weihl described a technique called *supertype abstraction* [20]. In supertype abstraction one uses a type system that statically determines a type for each expression. Each such static type should be a supertype of all possible values of that expression, as is common in OO languages. Then, one does verification of method calls using the specifications associated with the static type of each call's receiver object. Supertype abstraction is valid when each subtype is a behavioral subtype of all its supertypes [16, 19, 20, 23]. Furthermore, supertype abstraction is modular, since it does not require knowledge of all types in the program; instead, for soundness, it relies on each new type added to the program being a behavioral subtype of its supertypes. This avoids both the need for global knowledge of the program and the need for respecification and reverification.

However, most contemporary runtime assertion checkers for OO languages do not follow supertype abstraction precisely,[1] since when checking the pre- and postconditions of a method call, they use the specifications from the receiver's dynamic type. This is inconsistent with static verification tools and with supertype abstraction. Chalin's work pointed out the desires of practitioners for a uniform semantics between in static and dynamic checking tools [5]. It also noted the practical problems that can arise from inconsistencies between the semantics of assertions.

The reason that contemporary RAC compilers for OO languages suffer from this inconsistency is due to the way they implement runtime assertion checks for methods. At the beginning of each method's code, RAC compilers typically inject code to check the method's precondition. At the end of the method they also inject code to check the method's postcondition. This checking code is then run from within the method found at runtime, using the receiver's dynamic type, and so these runtime checks reflect the specifications associated with that dynamic type. We say that a RAC compiler that checks specifications based on dynamic types in this way is *overly-dynamic*. In Findler and Felleisen's work on contract soundness [12], such an overly-dynamic RAC is unsound, since it does not assign blame (issue error messages) properly.

Besides the inconsistency caused by this overly-dynamic checking, RAC compilers have additional problems that we discuss in detail in Section 4.

This paper has two main contributions. The first is the explicit recognition of the inconsistency problems that plague contemporary RAC compilers due to their overly-dynamic checking. Second, it contributes an improved and more consistent form of runtime assertion checking for OO programs. We introduce the concept of *client-aware interface specification checking* (CAISC), and show how it checks pre- and postconditions for OO programs in a way that respects supertype abstraction and information hiding. This paper is the first to explicitly identify and fix these problems related to runtime checking of OO programs.

Our CAISC technique works by injecting code to check pre- and postcondition checks into clients at the sites where they call methods. Because code is injected at the site of each method call, it can properly use the specifications associated with the receiver's static type.

[1] The only OO RAC tools that follow supertype abstraction are: Code Contracts [11], which only checks preconditions on the client-side and does not allow differences in preconditions for subtypes, and the checker for Racket, which follows the ideas in [12].

In this paper, we focus on formal interface specifications and runtime assertion checking for methods. Formal interface specifications include contracts written in Eiffel [24], JML [17], Spec# [2], and Code Contracts [11]. We use JML for concreteness, but the ideas we present could also be applied to Eiffel, Spec#, Code Contracts, or other interface specification languages. We prototype the notion of client-aware interface specification checking with our Aspect-JML RAC compiler (ajmlc) [31, 29].

The rest of the paper is organized as follows: Section 2 provides definitions and background. Section 3 presents the main example used in the paper. Section 4 surveys the problems of runtime checking for OO programs. Section 5 presents the key properties of client-aware interface specification checking. Section 6 analyzes the effectiveness of client-aware interface specification checking based on our findings and observations. Related work and open research issues are discussed in Section 7, and Section 8 summarizes our work.

2 Definitions

This section provides some definitions of important terms we use later.

2.1 Behavioral Subtyping

In essence, behavioral subtyping means that each type obeys the specifications of all of its supertypes. Subtypes must obey the specifications of their supertypes; because these supertype specifications are the ones used in reasoning with supertype abstraction [19]. This idea is reflected in the Liskov Substitution Principle, which was stated at Liskov's invited talk during OOPSLA 1987 [22, p. 25]:

> *"If for each object `o1` of type `S` there is an object `o2` of type `T` such that for all programs `P` defined in terms of `T`, the behavior of `P` is unchanged when `o1` is substituted for `o2` then `S` is a subtype of T."*

2.2 Specification Inheritance in JML

In JML, behavioral subtyping is enforced by specification inheritance. That is, a subtype inherits not only fields and methods from its supertypes, but also specifications such as pre- and postconditions. The way that method specifications are inherited ensures that each overriding method must satisfy the specifications of the methods it overrides [16].

To explain JML's method specification inheritance, let $T \rhd (\mathit{pre}, \mathit{post})$ denote an instance method specification written in a type T with precondition *pre* and postcondition *post* According to Leavens's definition [16, Definition 1], if $T \rhd (\mathit{pre}, \mathit{post})$ and $T' \rhd (\mathit{pre}', \mathit{post}')$ are specifications of an instance method m, and U is a subtype of both T and T', then the join of $(\mathit{pre}, \mathit{post})$ and $(\mathit{pre}', \mathit{post}')$ for type U, written $(\mathit{pre}, \mathit{post}) \bigsqcup^{U} (\mathit{pre}', \mathit{post}')$, is the specification $U \rhd (p, q)$, where the precondition p and postcondition q are respectively given by Formulas (1) and (2) below [16].

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

$$(\backslash\textbf{old}\,(pre) \Rightarrow post) \;\texttt{\&\&}\; (\backslash\textbf{old}\,(pre') \Rightarrow post') \tag{2}$$

This definition states that the precondition p is the disjunction of *pre* and *pre'*. The postcondition q is the conjunction of implications, such that when one of the preconditions holds in the pre-state, then the corresponding postcondition must hold.

2.3 Modularity and Inconsistency

We say that a specification language *supports separated specification* if specifications for a module can be written outside of its implementation's code (e.g., in a separate file).

A RAC and a static verification tool (or a verification logic) are *inconsistent* if for some executable method specification[2] and some call of that method, one of the following happens:

1. the static verification tool says that the call has a specification violation, and yet the RAC never detects that violation, or
2. the RAC detects a violation for that call, but the static verification tool says there is no violation.

The first kind of inconsistency is an incompleteness of the RAC, and the second is an unsoundness. The RACs we mentioned previously are sound, so we do not expect the second kind of inconsistency.[3] We will show an example of the first kind of inconsistency with respect to checking of preconditions, due to the overly-dynamic nature of contemporary RACs.

A RAC has *modular runtime checking* if it can dynamically check each method call based on the interface specifications associated with the receiver's static type. That is, to have modular runtime checking, a RAC should consistently apply the principle of supertype abstraction [20]. Note that in a specification language like JML that has specification inheritance, the specifications associated with a given static type will also include specifications inherited from that type's supertypes [16].

We combine these ideas in the following definition.

Definition 1. *Suppose a specification language has a standard static verification logic or static verification tool. Then a RAC for that specification language* supports modular reasoning *if it supports separated specification, is not inconsistent with the language's verification logic or a static verification tool, and has modular runtime checking.*

3 JML Background and Information Hiding

This section introduces a simple example that will be used throughout the paper. We use it in this section to explain the concepts of information hiding that our technique enforces.

[2] A method specification is *executable* if the RAC can always evaluate it. For example, in JML, some specifications that use quantifiers are not executable.

[3] The fact that contemporary RACs for OO languages are not unsound, but only incomplete, partly explains why these inconsistencies have been tolerated for many years.

3.1 Running Example in JML

Our running example is a figure editor system for editing graphical shape classes (e.g., points) [13, 15, 18, 30].

Figure 1 shows two JML specifications of this example: a protected specification for privileged clients (e.g., subclasses) on the left-hand-side, and a public specification on the right-hand-side. JML specifications are written in annotation comments[4], which start with an at-sign (`@`). In JML, a method specification may have several specification cases which are written before the method's header. The specification cases shown start with a visibility modifier and **`normal_behavior`**. The **`normal_behavior`** keyword indicates that the method must not throw exceptions when that specification case's precondition is satisfied; thus it must finish normally in a post-state that satisfies the corresponding postcondition. In JML, preconditions are introduced by the keyword **`requires`** and postconditions by **`ensures`**.

The JML specification cases for a method are separated by the keyword **`also`**, which represents the join operator defined by equations (1) and (2) (see Section 2). The meaning of **`also`** is thus that each specification case must be obeyed when its precondition holds [16]. Specifications of overridden methods start with **`also`** to remind the reader that the method must also obey all inherited specification cases. The JML notation `\`**`old`**`(_x+dx)` means the pre-state value of `_x+dx`. The invariants in JML are expressed by the keyword **`invariant`**. The invariants defined in this example restrict points to the upper right quadrant.

3.2 Information Hiding

Leavens and Müller [18] present rules for information hiding in specifications for Java-like languages. Normally, information hiding [26] controls which parts of a class are visible to clients (including subclasses). This aids maintenance because hidden implementation details can be changed without affecting clients. However, the benefits of information hiding apply also to other artifacts, such as documentation and specifications [18]. For example, one should keep invariants that describe implementation details hidden, so that when either is changed, clients are not affected.

Leavens and Müller's Rule 1 [18] says that specifications should not expose hidden class members. This implies that the protected invariant of class `Point` must not mention, for example, private fields.

In the same manner, a public client cannot use the protected specifications of class `Point` to check the correctness of calls to method `moveBy`, since they are not visible to such a public client [18, Rule 2]. However, clients must obey specifications that they can see. For example, the overridden method `moveBy` in the subclass `ScreenPoint` must refine the protected specification cases for `moveBy` that it inherits from `Point` [18, Rule 3]. Furthermore, such protected specification cases can be used to verify super-calls [32].

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

```
package p;                                 |package p;
interface Shape {}                         |public class Point2 implements Shape {
public class Point implements Shape {      |  //@ public model int x, y;
  protected int _x, _y;                    |  protected int _x, _y;
                                           |  //@ protected represents x = _x;
  /*@ protected invariant _x >= 0          |  //@ protected represents y = _y;
    @                  && _y >= 0; @*/     |
                                           |  /*@ public invariant _x >= 0
  /*@ protected normal_behavior            |    @               && _y >= 0; @*/
    @  requires _x + dx >= 0               |
    @        && _y + dy >= 0;              |  /*@ public normal_behavior
    @  ensures _x == \old(_x + dx)         |    @  requires x + dx >= 0
    @        && _y == \old(_y + dy);       |    @        && y + dy >= 0;
    @*/                                    |    @  ensures x == \old(x + dx)
  public void moveBy(int dx, int dy) {     |    @        && y == \old(y + dy);
    _x += dx;                              |    @*/
    _y += dy;                              |  public void moveBy(int dx, int dy) {
  }                                        |    // implementation like in class Point
}                                          |  }
package p;                                 |}
public class ScreenPoint extends Point {   |package p;
                                           |public class ScreenPoint2 extends Point2{
  /*@ also                                 |
    @ protected normal_behavior            |  /*@ also
    @  requires _x + dx < 0;               |    @ public normal_behavior
    @  ensures _x == 0;                    |    @  requires x + dx < 0;
    @ also                                 |    @  ensures x == 0;
    @ protected normal_behavior            |    @ also
    @  requires _y + dy < 0;               |    @ public normal_behavior
    @  ensures _y == 0;                    |    @  requires y + dy < 0;
    @*/                                    |    @  ensures y == 0;
  public void moveBy(int dx, int dy) {     |    @*/
    if(_x + dx >= 0) _x += dx;             |  public void moveBy(int dx, int dy) {
    else             _x = 0;               |    // implementation like in
    if(_y + dy >= 0) _y += dy;             |    // class ScreenPoint
    else             _y = 0;               |  }
  }                                        |}
}                                          |
```

Fig. 1. The protected and public specifications of the shape classes with JML.

On the other hand, invisible specifications need not be refined or obeyed. Thus overriding methods cannot be required to refine supertype specification cases that they cannot see. Similarly, since the protected precondition of method `moveBy` in class `Point` is not visible to public clients, so calls from such clients need not satisfy the `moveBy`'s protected precondition. Hence the example in the left-hand-side of Figure 1 is an unsatisfiable specification, because the public precondition (**`true`**) does not allow the implementation to satisfy the protected postcondition and invariant.

To prevent such unsatisfiable specifications, Leavens and Muller complement their rules with two guidelines [18]: (i) one should write at least one specification case with the same visibility as the method and (ii) one should make sure that the method's effective public precondition is at least as strong as the disjunction of the hidden preconditions.

Because of this, we should either change the visibility modifier of the method `Point.moveBy` to **`protected`** or write a public (client-visible) specification that can be respected by a public caller. We take the latter approach in the right-hand-side of Figure 1, where we define variants of classes `Point` and `ScreenPoint` named `Point2` and `ScreenPoint2`, respectively.

The classes `Point2` and `ScreenPoint2` are specified using model fields [7, 16, 21]. A model field is a specification-only field that is an abstraction of some concrete state. In the right-hand-side of Figure 1, the value of the public model fields `x` and `y` are determined directly by the values of the corresponding protected fields `_x` and `_y`. This relationship is specified by JML **`represents`** clauses. These **`represents`** clauses are protected, since they mention protected fields. This illustrates how model fields can be used to hide design details [7, 16, 21].

4 The Problems and their Importance

This section presents the key problems of the overly-dynamic effect of runtime assertion checking on OO programs in presence of subtyping and dynamic-dispatch. We discuss these problems in the following using the running example (in Figure 1).

4.1 Inconsistent Reasoning Problem

The first runtime assertion checking problem occurs when the specifications used to check the correctness of a method call are based on the receiver's dynamic type. The problem, as described in the introduction, is that such checking is inconsistent with supertype abstraction.

For example, consider the following client code that works with class `Point2` (of Figure 1):

```
package q; // public client
public class ClientClass {
  public void clientMeth(Point2 p) {
    //@ assume p.x == 0 && p.y == 0;
    p.moveBy(-1,-1);
  }
}
```

The technique of supertype abstraction uses the specification of the static type of the receiver `p` to reason about the method call to `moveBy`. Since `p`'s static type is `Point2`, supertype abstraction tell us to use the specifications given for `Point2`. Assuming that the point's x and y coordinates start at 0, this tells us that the call violates `Point2`'s public specification's precondition.

However, when using the JML runtime assertion checker (jmlc) [4, 6], we got no precondition violation when the receiver `p` has the dynamic type `ScreenPoint2`. That is, the effective precondition used to check the method call `p.moveBy(-1,-1)` is the specification given for type `Point2` joined with the specification cases of `p`'s dynamic type `ScreenPoint2`. This yields the effective precondition:

```
(x+dx >= 0 && y+dy >= 0) || ((x+dx < 0) || (y+dy < 0)).
```

Because this precondition allows negative arguments (the shadowed part), we got no precondition violation, contradicting `Point2`'s specifications. This overly-dynamic nature of RACs results in a common symptom that we call *masked precondition violation*.

For a programmer, this masked precondition violation is not quite intuitive without a proper understanding of the overly-dynamic execution techniques employed by the contemporary RACs. Hence, these RACS are inconsistent with supertype abstraction. In other words, the existing RACs check programs in a non-modular way.

This problem happens when the runtime precondition checks are instrumented on the supplier side. For instance, the JML RAC compiler (jmlc) [6], replaces the original method implementation with an assertion-checking method with the same name, into which checks for the pre- and postconditions that are injected. Thus, all client calls have pre- and postconditions checked on the supplier side. As a result, thanks to dynamic dispatch, method calls like `p.moveBy` include the specifications of subtypes, like `ScreenPoint2`.

Another way to look at this inconsistency problem is from the perspective of static verification tools. Consider again the method call `p.moveBy(-1,-1)`. If we use a static verification tool such as ESC/Java2 [4], we detect the expected precondition violation, based on the specifications of `p`'s static type `Point2`. This shows the modularity difference between such static verification tools and overly-dynamic RACs.

The overly-dynamic nature of contemporary RACs does not only affect preconditions; it also affects the runtime checking of other DbC features like postconditions and invariants. For example, suppose we strengthened the postcondition of the method `moveBy` in `ScreenPoint2`:

```
public class ScreenPoint2 extends Point2 {
  /*@ also
    @ public normal_behavior
    @  requires \same;
    @  ensures x <= 200 && y <= 200;
    @ also
    @ ...
    @*/
  public void moveBy(int dx, int dy) {
    // implementation like in class ScreenPoint
  }
}
```

The refined postcondition restricts the `Point2`'s coordinates to be less than or equal to 200. Note that the corresponding precondition for this refined postcondition is the same defined in `Point2`. Hence, we use the JML keyword **`\same`** to avoid duplicating the same precondition already defined in class `Point2`. Now consider the following client code:

```
package q; // public client
public class ClientClass {
  public void clientMeth(Point2 p) {
    //@ assume p.x == 0 && p.y == 0;
    p.moveBy(201,201);
  }
}
```

As mentioned, since `p`'s static type is Point2, we should use the specifications given in `Point2`. However, assuming that the receiver `p` has dynamic type `ScreenPoint2`, we get a postcondition violation for such method call. According to the Formula 2, the effective postcondition used to check the method call `p.moveBy(201,201)` was:

```
\old((x+dx >= 0 && y+dy >= 0))
  ==> (x == \old(x+dx) && y == (\old(y+dy)))
  && (x <= 200 && y <= 200)).
```

Since this postcondition requires the coordinates to be less than or equal to 200, the contemporary RACs like the jmlc reports a postcondition violation. In relation to the supertype abstraction, we should not consider the strengthened postcondition (the shadowed part) in reasoning about the method call `p.moveBy(201,201)`. So the RACs should not report any postcondition violation. We call this overly-dynamic symptom as *unexpected contract violation*. This term is more general since it comprehends any DbC feature that can be conjoined by conjunction such as postconditions and invariants.

Unexpected contract violation are inconsistencies (with respect to static verification tools) involving postconditions. With our technique of client-side RAC, such inconsistencies disappear.

4.2 Visibility Rules Checking Problem

As discussed above, Leavens and Müller's rules for information hiding in specifications [18] restrict proof obligations on method calls so that reasoning about such calls can only use specifications that are visible at the call site.

Recall that the right-hand-side of Figure 1 is written with public specifications, which are visible to all clients. Suppose now that we change the visibility modifier of method `moveBy`'s specification cases in class `ScreenPoint2` to be **`protected`**[5] as we had before with class `ScreenPoint`. Suppose further that we have the following public client code of this modified version of `ScreenPoint2`:

```
package q; // public client
public class ClientClass {
  public void clientMeth(SceenPoint2 p) {
    //@ assume p.x == 0 && p.y == 0;
    p.moveBy(-1,-1);
  }
}
```

In this scenario, according to supertype abstraction, `ScreenPoint2`'s specification must be used to reason about the correctness of such a call. However, in this scenario the specification cases of `ScreenPoint2` are not visible at the call site, since we are assuming that they have been made **`protected`**. Thus the only specification visible for `moveBy` is the public specification case inherited from class `Point2`. As a result, based on supertype abstraction and these information hiding rules, in this scenario there should be a precondition violation for the call to `moveBy`.

However, the supplier-side checking code generated by existing RACs ignores visibility modifiers in specifications, which is inconsistent with the proper way that the call in this scenario should be checked. For instance, if we use jmlc [4] in this scenario, it

[5] As discussed in our running example section, making the specification cases more hidden in method `moveBy` of class `ScreenPoint2` is valid because the Leavens and Müller's Rule 2 [18] and guidelines are respected. Note that the method `ScreenPoint2.moveBy` has a public specification case which is inherited from its supertype denoted by `Point2`.

does not detect any precondition violation. This happens because the protected specification cases, which are not visible to public clients, are used during runtime checking

In summary, due to the supplier-side instrumentation approach adopted by RACs, all specifications with different visibility levels are checked without respecting the information hiding rules [18]. In this case, when a method call occurs, the runtime checks in the supplier class do not take into account what kind of client (i.e., privileged or non-privileged) called the method. This is inconsistent with proper static verification of JML client code for such calls.[6]

4.3 Library Runtime Checking Problem

Nowadays there is large-scale reuse of components, due to standardization of libraries and frameworks in popular programming languages such as C++, Java, and C#. Yet most libraries and frameworks are only specified informally using natural language [27]. However, the use of natural language as the primary documentation technology can cause problems [27]:

> *"The information must be precise, i.e. have only one possible interpretation. When reader and writer, (or two readers) can interpret a document differently, compatibility problems are very likely. Experience shows that it is very hard to write unambiguous documents in any natural language."*

Besides the problem of specification for such libraries, for a RAC compiler the problem is how to check calls to library methods. Contemporary RAC compiler (e.g., jmlc [4]) need the source code for libraries, in order to inject checking code into the library classes (on the supplier side). This makes it impossible for them to check client code for such libraries at runtime.

5 Client-Aware Interface Specification Checking (CAISC)

To avoid the problems described above, one needs to avoid the overly-dynamic nature of contemporary RAC compilers, which results from their checking method specifications on the supplier side. By contrast, our approach, which we call *client-aware interface specification checking*, or CAISC, uses client-side checking to avoids these problems.

To give an example, consider Figure 2, which contains code for the type `Point2` (lines 4-13) and a separated specification for it (lines 14-32). This separated specification would be contained in a file `Point2.jml`. The figure also shows a public client (lines 33-50).

In a program logic, CAISC is embodied in the proof rule for method calls, which allows us to derive $\{P\}\, p.m(\boldsymbol{a})\, \{Q\}$ only from a specification $(T \rhd pre^T_m,\, post^T_m)$ associated with the static type T for the receiver p. Usually, an automated verifier uses weakest precondition semantics and achieves modularity by replacing a call $p.m(\boldsymbol{a})$

[6] Unfortunately, none of existing JML static verification tools properly check these visibility rules [18] either.

```
01 package p;
02 interface Shape {}
03 /** implementation (.java) */
04 public class Point2
05             implements Shape{
06  protected int _x, _y;
07
08  public void moveBy(int dx,
09                     int dy){
10   _x += dx;
11   _y += dy;
12  }
13 }
```

```
14 package p;
15 /** interface specifications (.jml) */
16 public class Point2 implements Shape{
17   //@ public model int x, y;
18   protected int _x, _y;
19   //@ protected represents x = _x;
20   //@ protected represents y = _y;
21
22   /*@ public invariant x >= 0
23     @                 && y >= 0; @*/
24
25   /*@ public normal_behavior
26     @  requires x + dx >= 0
27     @        && y + dy >= 0;
28     @  ensures x == \old(x + dx)
29     @       && y == \old(y + dy);
30     @*/
31   public void moveBy(int dx, int dy);
32   }
```

```
33 package q; // public client
34 public class ClientClass1 {
35   /** client-side reasoning */
36   /** and instrumentation */
37   public void clientMeth1(Point2 p) {
38     /*@ assume p.x == 0
39             && p.y == 0; @*/
40
41     /*@ assert p.x + -1 >= 0
42       @      && p.y + -1 >= 0; @*/
43     p.moveBy(-1,-1);
44     /*@ assume p.x == \old(p.x + -1)
45       @ && p.y == \old(p.y + -1); @*/
46
47     /*@ assume p.x >= 0
48       @      && p.y >= 0; @*/
49   }
50 }
```

Fig. 2. Example for Client-Aware Interface Specification Checking.

by the sequence of "**assert** $pre_m^T[\boldsymbol{a}/\boldsymbol{f}]$; **assume** $post_m^T[\boldsymbol{a}/\boldsymbol{f}]$" [1]. (The notation $[\boldsymbol{a}/\boldsymbol{f}]$ means the substitution of the actual parameters $\boldsymbol{a}$ for the formals $\boldsymbol{f}$.)

In our example, the reasoning as well as the instrumentation of the public client can be observed on lines `38-48` of Figure 2. In runtime assertion checking, all **assume** statements are checked, just like **assert** statements. Thus we can illustrate the runtime checks as a sequence of **assert** and **assume** statements. The resulting code is the right way to reason about the dynamically-dispatched method call `p.moveBy(-1,-1)` using supertype abstraction. That is, we inject runtime checks at the site of each method call to check the pre- and postconditions of the statically visible specifications for the call.

5.1 Design Decisions for CAISC

Tool support is necessary to automatically instrument and check the specifications in CAISC.

We added support for CAISC to the Aspect-JML RAC compiler (ajmlc) [31, 29] which is available online at `http://www.cin.ufpe.br/~hemr/JMLAOP/ajmlc.htm`. This is the first RAC to support CAISC. This section discusses the key design decisions in our implementation.

Information-Hiding Specifications. One decision is to use all the information hiding facilities of JML, including privacy modifiers for specifications. While this leads to more complex specifications and forces the runtime assertion checking to be aware of such privacy modifiers, it has benefits for information hiding.

Separated Specification. A second decision is to allow specifications to be separated from program code. While this complicates the RAC, which must find and use these extra specification files, it allows the specification of proprietary libraries and frameworks.

Open Classes Support. To support separated specifications we follow JML in allowing a programmer to add new members (e.g., methods or fields) to existing classes without modifying existing code.[7] These added members can be used to enhance the expressive power of the behavioral specifications. For example, in Figure 2 we show how to add new public model fields and use them in public specifications.

An implementation challenge is how to support this open class feature even if the source code is not available. For this our prototype implementation relies on the inter-type declaration feature of AspectJ [13].

Compilation and Code Injection. We decided to allow specifications to be added or modified whenever needed. This implies that either client side checks redirect to a central place where assertion checking code is stored, or that all client code must be recompiled with the RAC compiler whenever a type's specifications change. However, because our prototype implementation uses AspectJ to inject code, the generated aspects are a central place where assertion checking code is stored. Furthermore, AspectJ's weaver injects into the client code only what are essentially calls to the generated assertion checking code at client call sites. Indeed AspectJ's weaver could reweave a program without the need to recompile all client code when a specification changes.

Full client assertion checking. In order to investigate the power of CAISC, our prototype implementation does no supplier-side assertion checking. That is, all specifications are only checked on the client side. This means that only visible specifications are checked at the sites of client method calls. It also implies that the implementation supports modular reasoning.

Implementation Strategy. It is often claimed in the literature that the contracts of a system are de-facto a crosscutting concern and fare better when modularized with aspect-oriented programming (AOP) [14, 13] mechanisms such as pointcuts and advice [13, 3, 30]. Generative programming is another research area in which AOP has been successfully used [10]. The ajmlc RAC combines the best of both of these worlds by generating an AspectJ program [31, 29]. Hence, we decided to adapt ajmlc to implement CAISC. In ajmlc, the generated AspectJ advice for checking pre- and postconditions is compiled through an AspectJ weaver (e.g., ajc), which weaves them with the standard Java code or jar files if the source code is not available.

6 Analysis

In this section we present an analysis of the benefits of runtime checking with CAISC. The analysis is broken into two parts: (i) first we compare our approach, using our definitions (see Section 2), against other interface specifications languages using the running example (Section 3), and (ii) we describe some preliminary experience with checking a real system. Detailed results are available from [28].

[7] This functionality originated in Flavors [25] where Flavor declarations did not lexically contain their methods. The term open class is due to [9]. In JML such added features are called "model" fields and methods.

Table 1. Analysis of non-CAISC and CAISC-based interface specifications of shape classes.

		Separated	Privacy	Consistency	Modular Reasoning
non CAISC	Eiffel	no(1)	medium(2)	medium(3)	no
	JML/jmlc	no(1)	high	medium(3)	no
	Spec#	no(1)	medium(2)	medium(3)	no
	Code Contracts	no(1)	medium(2)	medium(3)	no
CAISC	JML/ajmlc	high	high	high	yes

(1) `Shape` classes are contaminated with scattered and tangled behavioral specifications.
(2) The privacy of specifications are not a documented part of the interface and/or they are not automatically enforced.
(3) Because checking does not respect supertype abstraction or privacy modifiers.

6.1 Analysis of RAC Compilers

We first analyzed the non-CAISC and CAISC-based RACs according to our definitions from Section 2 using the shapes example (see Section 3). This analysis is summarized in Table 1. Note that in Table 1 the column labeled "Supports Modular Reasoning" summarizes our judgment related to our definition of what it means for a RAC to support modular reasoning. For this comparative study, we considered the RAC tools of Eiffel [24], JML (i.e., jmlc [6]), Spec# [2], and Code Contracts [11].

The Supplier-Side Approach With RAC compilers that use the supplier-side approach, the runtime checking of the `Shape` classes (e.g., `Point`) fails to satisfy our criteria:

- they are not textually separated from the implementation code,
- they do not properly check privacy specifications, and
- due to the overly-dynamic supplier-side checking subtyping and dynamic-dispatch, these tools are inconsistent with supertype abstraction, and so do not support modular reasoning according to our definition.

The Client-Side Approach The ajmlc tool, which we modified to use CAISC meets our criteria for modular reasoning for the `Shape` classes:

+ it works with textually-separated specifications,
+ it properly checks specifications with privacy modifiers, and
+ due to the CAISC approach, it is consistent with supertype abstraction and thus supports modular reasoning according to our definition.

6.2 The Health Watcher Case Study

We also validated our client-aware interface specification checking through a medium-sized case study. The chosen system was a real web-based information system, called Health Watcher (HW) [33]. The main purpose of the HW system is to allow citizens to register complaints regarding health issues. This system was selected because it has a detailed requirements document available [33]. This requirements document describes 11 use cases.

We analyzed the runtime checking for the HW system, comparing the non-CAISC and CAISC ajmlc-based. The JML specifications used were for 5 of the 11 HW use cases. Then we monitored these use cases of the HW system at runtime. We are able to confirm that the benefits of CAISC were obtained in this case study. Our results is available online at [28].

Understanding the Overly-Dynamic Structure of HW. One of the most important issues is to recognize how the overly-dynamic checking done by contemporary runtime assertion checkers works, and how it harms the runtime verification of real systems like HW. We now analyze the aforementioned symptoms of the overly-dynamic nature of RACs called such as masked precondition violation.

In relation to masked precondition violation symptom, let us consider the "search employee" use case. The HW's requirements document says that the actor employee must provide a valid login and password in order to successfully try to login in the HW system. In the requirements document, a valid login and/or password should be neither empty nor blank. Hence, we provide the following JML specification for the HW facade interface:

```
//@  requires login != null;
//@  requires !login.equals("");
//@  requires !login.equals(" ");
//@  ensures \result.getPassword() != null;
//@  ensures !\result.getPassword().equals("");
//@  ensures !\result.getPassword().equals(" ");
public Employee searchEmployee(String login) {...}
```

However, even though an employee user passes a blank login string, the error recovery (exception handling) of the HW system asks the employee user to try again since blank logins and/or passwords are not allowed and therefore not persisted in the system. As such, in the `HealthWatcherFacade` class (which is a subtype of `IFacade` interface), there is an added specification case with weaker precondition:

```
//@ also
//@  requires login.equals(" ");
//@  ...
public Employee searchEmployee(String login) {...}
```

This JML specification case says that one can now pass blank logins to the system. In the HW servlet denoted by the type `ServletLogin` responsible for the searchEmployee (login) use case, there is the following method call:

```
Employee employee = facade.searchEmployee(login);
```

The static type of `facade` is `IFacade`. Hence, according to the supertype abstraction technique, we need to look at the specifications associated with `IFacade` to reason about the such a method call. Suppose the variable `login` has a blank string, thus by using the specifications of the method `searchEmployee` in `IFacade` we should have a precondition violation. However, since the runtime type of `facade` is `HealthWatcherFacade`, the non-CAISC runtime assertion checkers consider a weaker precondition instead, resulting in no precondition violation, i.e., a masked precondition violation occurs. In the HW system we can only detect the precondition violation (in conformance with the specifications of type `IFacade`) when we employ our CAISC approach. In the following we show the precondition violation generated by our ajmlc compiler, which adopts the CAISC approach:

```
> ajmlrac
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLInternalPublicPreconditionError:
by method:
healthwatcher.view.IFacade.searchEmployee regarding specifications at
File "healthwatcher.view.IFacade.java",
[spec_case]:
line 113, character 27, when
  ’login’ is ’ ’
... more
```

More Discussions. First, we found a bug in the "insert employee" use case. The code snippet below was extracted from the business layer. It illustrates the piece of code of the `IFacade` class responsible for implementing the "Insert Employee" use case.

```
1 //@ requires !employee.getLogin().equals("");
2 //@ requires !employee.getName().equals("");
3 //@ requires !employee.getPassword().equals("");
4 //@ requires !employee.getLogin().equals(" ");
5 //@ requires !employee.getName().equals( );
6 //@ requires !employee.getPassword().equals(" ");
7 public void insert(Employee employee) {...}
```

Note that in order to successfully insert a new employee in such a system, a user name, login, and password should be provided. Any missing employee information violates the use case requirement. Hence, in order to prevent any malicious client from bypassing such a validation (leading to inconsistent data in the system), 6 basic preconditions (lines 1-6) are added to the `insert` method from the Business layer (line 7).

Hence, any attempt to insert a new employee with missing required data, should result in a precondition violation that should be presented to the user. The key point is that in the CAISC approach we had a precondition violation pointing out the visibility of that precondition. This is useful because, unlike the standard approaches (non-CAISC), we can now identify what kind of clients are violating a precondition. Consider we have the method call `f.insert(`**`new`**` Employee("login","","pwd"));` in which `f`'s static type is `IFacade`. This method call passes an invalid name (empty in this case), according by the precondition described on line 2. As a result, by using our CAISC approach, we got the following precondition error:

```
> ajmlrac
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLInternalPublicPreconditionError:
by method:
healthwatcher.model.employee.Employee.<init> regarding specifications at
File "healthwatcher.model.employee.Employee.java",
[spec_case]:
line 28, character 39
(healthwatcher.model.employee.Employee.java:28),
... more
```

This error message says that we had a precondition violation in the constructor of class `Employee`. This is due to the constructor call passed as argument to create a new

`Employee` object. Since we have the same preconditions for the `Employee`'s constructor, we got this precondition violation first. We can have a precondition error directly associated with the method `insert` if we turn off the preconditions of the class `Employee`. The main point here is that we can now blame the public client that made the invalid call to the `Employee`'s constructor.

Second, for this particular system, we observed no significant difference in the final bytecode size between the bytecode generated by both non-CAISC and CAISC versions of our ajmlc tools. The reason for this is that AsepctJ only injects a small amount of code at each client call site; the injected code just calls the advice that checks assertions, and so the overhead is fairly small. The code to check pre- and postconditions is not repeated at each client site, but is centralized in the generated aspect.

Third, if one want to make a library of the HW system, one can only be benefited of runtime checking when employing our CAISC approach; otherwise, since the source could is missing for third party applications, we can neither specify nor check specifications of the HW library. If we look at the HW's source code, this is particular true for the util package that the HW system contains. In other words, one can make this util package available as a library to other projects besides the HW. In this case the CAISC is suitable to distribute the library with its interface specifications that can be checked during runtime.

The last but not least, we realize that the HW complaints are implemented by means of a hierarchy. In addition, in the HW servlets (e.g., `ServletInsertFoodComplaint`) responsible for including complaints in the system are based on the `Complaint` type, which is the supertype of all specialized complaints of the system (e.g., food complaint). Hence, if we are using runtime assertion checking without CAISC, the more kinds of complaints we add in the system compromise the overall modular reasoning of the method `insertComplaint`, contained in the HW's `IFacade` that is called within those HW servlets (which handles the HW complaints). This happens because we need to extend our case analysis to include the new kinds of complaints. With CAISC we avoid any attempt to perform a case analysis to reason about the method calls.

7 Discussion

7.1 Tackling the Runtime Checking Problems

Since CAISC is based on the static type of the receiver of a particular method call, we can exploit all the modularity benefits of supertype abstraction during runtime checking, which solves the inconsistency problem. Consistency with static verification tools is beneficial in that it is less confusing to users. In addition, CAISC respects privacy modifiers in interface specifications, solving the visibility rules checking problem.

Finally, our implementation of CAISC supports separated specifications and even allows model fields and model methods to be specified in separate specification files. This is a great help in solving the library runtime checking problem, since the specifier does not need to modify library sources or byte codes to do runtime assertion checking with ajmlc.

7.2 Benefits of Using AspectJ as a Back-End

Since our approach is based on client-side instrumentation, the more clients a program has, the more instrumentation code will be generated and added to the program. This is a small overhead, compared to supplier-side instrumentation, where the runtime checks are generated only in a single place (in the supplier class). However, due to ajmlc's use of AspectJ, most of the code that it generates goes into a single place, namely an aspect. As mentioned, only a small amount of overhead was observed in the HW system (compared to jmlc).

Similarly, the use of AspectJ avoids the need to recompile client code when new clients are added to a program or when specifications change, as all that is needed is for AspectJ to reweave any changed specifications, a process that can be done as late as load time.

7.3 Other Interface Specification Languages and Related Work

As mentioned in the introduction, besides JML [17], there are several other interface specification languages, such as Eiffel [24], Spec# [2], and Code Contracts [11]. The latter two are both interface specification languages for the C#.

Eiffel has the same runtime checking problems discussed here with JML's jmlc tool (since the implmentation of jmlc drew heavily on techniques pioneered by Eiffel).

In Spec# and Code Contracts (as in Eiffel and JML), a method's specification is inherited by its overriding methods. However, Spec# and Code Contracts do not allow a programmer to changes the precondition of an overriding method. A method override can only add more (stronger) postconditions or invariants. The authors of Spec# and Code Contracts say that callers (clients) expect the specification of the static resolution of the method call to agree with runtime checking, which is what we have called consistency with supertype abstraction. However, their design limits what one can specify for subtypes and only works for preconditions.

Code Contracts has another interesting relationship to our work: it supports runtime checking at call sites. However, in Code Contracts only preconditions can be checked at call sites. With CAISC, we can also check postconditions and invariants at call sites. In addition, Code Contracts do not support open classes, information hiding in separated specifications, and private members cannot be mentioned in specifications (e.g., postconditions) when specifications are separated from the source code. As a consequence, it would not be able to properly check postconditions with different privacy modifiers.[8]

The work on Spec# and Code Contracts points out that the major differences between RAC compilers has to do with their treatment of preconditions, since precondition checking is the main way that users can observe inconsistencies between runtime and static checking. However, our CAISC approach is more general, because it also allows checking of postconditions (and other specifications, such as invariants) for libraries when source code is not available.

[8] Code contracts does not support privacy modifiers at all at present. But if they were added, then the ability of Code Contracts to do client-side checking only for preconditions would make it possible for the tool to respect privacy modifiers only for preconditions.

Findler and Felleisen's work [12] describes contract soundness for a language called Contract Java. In Contract Java a programmer can specify pre- and postconditions of methods defined in classes and interfaces. Their work uses supertype abstraction to do runtime checking of pre- and postconditions for method calls — using specifications associated with the static type of the receiver. Their translation rules, which inject runtime checking code, check pre- and postconditions associated with the static types at call sites (client-side checking). Hence their work is a precedent for runtime checking for violations of supertype abstraction, and for our work on CAISC.

However, Findler and Felleisen's work has some limitations when compared to our CAISC. First, their work neither considers separate specifications for different privacy levels nor enforces information hiding of specifications, as we do. Furthermore, our client-side checking can check other JML DbC features at call sites (e.g., invariants, constraints, and model specifications), while their work only mentions and provides rules for pre- and postconditions. Findler and Felleisen's work also has a different focus than our work, as they are primarily concerned with enforcing behavioral subtyping and presenting the novel idea of soundness of contract checking. Thus, unlike our work, they do not explicitly point out the problems that can occur if a runtime assertion checking tool is over-dynamic and does not obey supertype abstraction. Their contract checking mechanism has two steps: (1) checking pre- and postconditions according to the receiver's static type, and (2) checking the dynamic type of the receiver for properly obeying Liskov and Wing's [23] conditions for behavior subtping, in order to detect *hierarchy violations*. To detect such hierarchy violations, this second check compares the values of the (self-contained) pre- and postconditions of the dynamic type's method, m, against the values of the contracts for all methods that m overrides in supertypes. One difference is that JML does not presume that method contracts are complete and self-contained, since it joins contracts with those of supertypes using **`also`**; thus in JML every subtype is a behavioral subtype [16], so no hierarchy violations are possible. A second difference is that we consider Findler and Felleisen's checks for hierarchy violations to be overly-dynamic, since they involve the dynamic type's specification.

7.4 Open Issues

Our evaluation of CAISC is limited to two systems, Health Watcher [33] and shape classes [15, 18, 30]. Larger-scale validation is necessary to analyze more carefully the benefits and drawbacks of CAISC. Library specification and runtime checking studies are one interesting open challenge. Another open issue is related to IDE support. We expect that IDE support could aid programmers by showing, for example, the scope of a precondition's applicability at the client-side site of a method call. Tools like AJDT already offer similar functionality for AspectJ programs that indicate which advice apply in a certain join point [15].

The interfaces we describe are the client-aware versions of standard Java classes and interfaces. More sophisticated interface technologies have been developed for OOP. One of these is the aspect-aware interfaces for AspectJ [15]. Our belief is that the basic idea of CAISC can be enhanced with some principles behind aspect-aware interfaces. Hence, comparison with other advanced interface technologies will be also useful to extend or refine this work.

8 Summary

Client-aware interface specification checking, CAISC, enables consistent modular runtime checking and modular reasoning in the presence of subtyping, dynamic dispatch, and information hiding in specfications. CAISC represents a change in the way that runtime checks are injected into code, since checks are placed in client code as opposed to being only done in supplier classes. Furthermore, by using AspectJ as a back-end, our tool is able to carry out this approach even for interfaces that are textually separated from source code. Since runtime checks are injected at the point of method calls, the client's perspective on the called method can be taken into account, which allows runtime checking to be consistent with supertype abstraction and privacy modifiers used for information hiding.

Acknowledgments

We thank Rustan Leino, Mike Barnett, Peter Müller, Shuvendu Lahiri, and Tom Ball for discussions about these topics. Mike Barnett also gave us a demo of Code Contracts.

We also thank Mira Mezini for detailed discussions about this paper while she visited us in Brazil.

Thanks to SBLP (Brazilian Symposium on Programming Languages) for bringing all the authors together before the ECOOP deadline.

This work has been partially supported by FACEPE under grant No. IBPG-1664-1.03/08 for Henrique Rebêlo. Ricardo Lima is also supported by CNPq under grant No. 314539/2009-3. The work of Leavens was partially supported by a US National Science Foundation grant, CCF-10-17262.

Online Appendix

We invite researchers to replicate our case study. Source code of the non-CAISC and CAISC versions of the running example and HW system, versions of the interface specification languages used, and our results are available at [28].

References

1. Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31:82–87, September 2005.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*. Springer-Verlag, 2005.
3. Lionel C. Briand, W. J. Dzidek, and Yvan Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.

4. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7:212–232, June 2005.
5. Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *International Conference on Software Engineering (ICSE)*, pages 23–33, Los Alamitos, California, May 2007. IEEE.
6. Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, 2002*, pages 322–328, 2002.
7. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6), 2005.
8. Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31:25–37, May 2006.
9. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. volume 35, pages 130–145, New York, NY, USA, October 2000. ACM.
10. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
11. Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10. ACM, 2010.
12. Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM.
13. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44:59–65, October 2001.
14. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.
15. Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.
16. Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*. Springer-Verlag, 2006.
17. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.
18. Gary T. Leavens and Peter Muller. Information hiding and visibility in interface specifications. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007.
19. Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006.
20. Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.
21. K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130, New York, NY, March 2006. Springer-Verlag.

22. Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.
23. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
24. Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
25. David A. Moon. Object-oriented programming with flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA)*, 1986.
26. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.
27. David Lorge Parnas. Precise documentation: The key to better software. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer Berlin Heidelberg, 2011.
28. Henrique Rebêlo, Gary T. Leavens, and Ricardo Lima. Modular enforcement of supertype abstraction and information hiding with client-side checking. Available from: `http://cin.ufpe.br/~hemr/ecoop12`.
29. Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Gary T. Leavens, Alexandre Mota, and César Oliveira. Optimizing JML feature compilation in ajmlc using aspect-oriented refactorings. Technical Report CS-TR-09-05, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, April 2009.
30. Henrique Rebêlo, Ricardo Lima, and Gary T. Leavens. Modular contracts with procedures, annotations, pointcuts and advice. In *SBLP '11: Proceedings of the 2011 Brazilian Symposium on Programming Languages*. Brazilian Computer Society, 2011.
31. Henrique Rebêlo, Sérgio Soares, Ricardo Lima, Leopoldo Ferreira, and Márcio Cornélio. Implementing java modeling language contracts with aspectj. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08. ACM, 2008.
32. Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 208–228, New York, NY, USA, 2000. ACM.
33. Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In *Proceedings of the 17th conference on Object-oriented programming (OOPSLA), systems, languages, and applications*, 2002.