

A Contextual Interpretation of Undefinedness for Runtime Assertion Checking

Yoonsik Cheon and Gary T. Leavens

TR #05-10

March 2005; revised July 2005

Keywords: undefinedness, runtime assertion checking, formal methods, exceptions, partial functions, JML language.

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); 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, monitors, test data generators, testing tools, theory; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

To appear in *Sixth International Symposium on Automated and Analysis-Driven Debugging (AADEBUG 2005)*, Monterey, California, September 19-21, 2005.

© ACM, 2005. This is author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in AADEBUG 2005.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

A Contextual Interpretation of Undefinedness for Runtime Assertion Checking

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
500 W. University Avenue
El Paso, Texas 79968-0518
cheon@cs.utep.edu

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

ABSTRACT

Runtime assertion checkers and static checking and verification tools must all cope with the well-known undefinedness problem of logic. This problem is particularly severe for runtime assertion checkers, since, in addition to the possibility of exceptions and errors, runtime assertion checkers must cope with non-executable expressions (such as certain quantified expressions). This paper describes how the runtime assertion checker of the Java Modeling Language (JML) copes with undefinedness. JML is interesting because it attempts to satisfy the needs of a wide range of tools; besides runtime assertion checking, these include static checking tools (like ESC/Java) and static verification tools. These other tools use theorem provers that are based on standard (two-valued) logic and hence use the underspecified total functions semantics for assertions. That semantics validates all the rules of standard logic by substituting an arbitrary value of the appropriate type for each undefined subexpression. JML's runtime assertion checker implements this semantics, and also deals with non-executable expressions, in a way that is both simple and practical. The technique implemented selects a value for undefined subexpressions depending on the context in which the undefinedness occurs. This technique enables JML's runtime assertion checker to be consistent with the other JML tools and to fulfill its role as a practical and effective means of debugging both code and specifications.

Categories and Subject Descriptors

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AADEBUD'05, September 19–21, 2005, Monterey, California, USA.
Copyright 2005 ACM 1-59593-050-7/05/0009 ...\$5.00.

General Terms

Documentation, Languages, Verification.

Keywords

Undefinedness, runtime assertion checking, formal methods, exceptions, partial functions, JML language.

1. INTRODUCTION

1.1 DBC and Assertions

Design by contract (DBC) [25, 26, 27, 28] is a technique for isolating interface errors in programs. In DBC, a module (such as a class in an object-oriented language) and its clients agree on a contract that specifies the details of the module's interface. The contract specifies obligations and rights for both the clients and the implementor. The most important kind of obligations for the clients are method *pre-conditions*, which state when clients can call each method. The most important kind of obligations for the implementor are method *post-conditions*, which describe the relation between the pre-states of such calls and the corresponding (possible) post-state(s). Thus method calls for which the pre-condition does not hold are errors in the client code, and calls for which post-condition does not hold are errors in the method's implementation.

In addition to pre- and post-conditions, several other kinds of Boolean-valued expressions, i.e., *assertions*, are used in DBC. These include class invariants and in-line assertions. Class invariants must hold in all visible states (both pre- and post-states of public methods, for example) and are often used to catch errors in an implementation's treatment of data structures. In-line assertions such as `assert` statements and loop invariants must hold whenever encountered during execution, and are useful for isolating errors within a method to a particular sub-section of the method's code.

1.2 Tools that Use Assertions

A wide range of tools can use assertions for finding and isolating errors in programs. The most common are runtime assertion checkers. Eiffel [27] and APP [28] are standard examples, but there are many other runtime assertion checkers available [1, 8, 19, 30]. All detect assertion violations dynamically at runtime. Such violations can be used to assign blame to subsections of a program, thereby aiding debugging

by preventing such bugs from propagating farther before being detected.

In addition to runtime assertion checkers, other tools can use assertions for debugging or reasoning about programs. One example is a unit testing tool [7, 9], which uses assertions to decide the success or failure of unit tests. Another well-known example is the ESC/Java tool [12]. ESC/Java uses static analysis to automatically find bugs, such as dereferencing null pointers and indexing arrays outside their bounds. In ESC/Java, a warning about such a problem that is due to not having an assumption about a method’s formal parameter can be suppressed by writing a pre-condition for that method; this pre-condition would then be checked automatically. Hence writing assertions can help users isolate bugs at their sources and has a nice side-effect of precisely documenting interfaces. Finally, static verification tools, such as the LOOP tool [14], can be used to prove that a (debugged) program is correct.

1.3 JML’s Semantics for Undefinedness

The Java Modeling Language (JML) is a formal specification language that can be used as a DBC tool. However, unlike most DBC languages, it aims to be a notation that can be used by many different tools [4], and in particular by both runtime assertion checking tools and tools that use static analysis or theorem proving [22]. All of the tools mentioned in the preceding paragraph work with JML.

Because JML caters to both static and dynamic tools it uses a somewhat different semantics for assertions. Unlike most DBC tools, JML does not simply propagate exceptions that occur in assertion evaluation to the user. However, like other DBC tools, JML’s runtime assertion checker must deal with undefined subexpressions that arise when evaluating assertions. This is because assertions, which in JML and other DBC tools are written in a subset of some programming language, may throw exceptions or encounter runtime errors. In logic, subexpressions that encounter such problems are said to have *undefined* values.

The undefinedness problem has been studied by many logicians (see [15, 29] for overviews). One standard solution is to use a three-valued logic. Three-valued logics use a third logical value, \perp , in addition to true and false [2, 3, 16, 18]. Such logics correspond to the approach used in many DBC tools [1, 19, 26, 30], since exceptions that arise during evaluation of assertions are propagated to the user, and hence can be thought of as having the value \perp .

One shortcoming of the three-valued logic approach is that one cannot use all standard rules of logic to reason about assertions. For example, consider the Java expression

`(a.length > 0) & (a != null)`

where `a` is an array variable. If `a` is `null`, then the subexpression `a.length` will throw a null pointer exception. This will happen despite the fact that `a != null` will be false when `a` is `null`, and thus the overall assertion would seem to have a reasonable truth value, in a two-valued logic. However, in a three-valued logic, this assertion has value \perp , as $\perp \& \text{false} \equiv \perp$. To avoid such problems, one has to write “protective” assertions [23]. For example, one would rewrite the assertion above as `(a != null) && (a.length > 0)`, which protects the subexpression `a.length` from undefinedness by evaluating `a != null` first, and by using the short-circuit operator `&&`.

It is inconvenient to reason using with three-valued log-

ics, since one must constantly consider the possibility that an expression has \perp as a result, and because classical logical rules, such as the law of the excluded middle, do not hold [13]. For this reason, PVS and other theorem provers use the semantics of modeling partial functions as *underspecified total functions* [13]. In this semantics, when a function, f , is applied to a value, v , $f(v)$ always returns some particular value of f ’s result type, but in some cases nothing may be specified about the exact value of $f(v)$; hence all functions are assumed to be total on their domains, but not all functions are completely specified on all inputs. This semantics yields a two-valued logic, since all Boolean-valued expressions are either true or false. Because it is two-valued, this semantics validates the classical laws of logic, keeping the logic simple and supporting calculational reasoning. Since the underspecified total functions semantics is used in important theorem provers (Simplify and PVS) on which some JML tools are based, it forms part of JML’s semantics for assertions [22, Section 2.7].

However, in JML undefinedness can arise for another reason not typically considered by logicians or runtime assertion checkers. To allow users to escape from formality, and to allow tuning the level of formality of JML specifications, JML features informal descriptions, such as `(* s is printed *)`, which are not executable [20]. Furthermore, since JML supports a wide variety of tools, it also contains some kinds of expressions, such as some forms of quantifiers, that the runtime assertion checker cannot execute. Most DBC tools do not have to deal with non-executable constructs, because their notation does not allow them.

JML’s semantics distinguishes these two different kinds of undefinedness. Consider a subexpression E_1 . We say that E_1 exhibits *angelic undefinedness* if E_1 cannot be handled by the tool (e.g. if it is not executable), and it exhibits *demonic undefinedness* if it can be handled but is otherwise undefined (e.g., due to an exception). A JML tool must obey the underspecified total function semantics for all occurrences of demonic undefinedness, and it cannot falsify an assertion solely due to angelic undefinedness. Since the value of an angelic undefinedness is unknown, tools cannot prove that assertions are false based on such angelically undefined subexpressions. For example, if an entire assertion is not executable (e.g., it is an informal description), then it must be considered to be true.

JML’s runtime assertion checker must follow this semantics when evaluating assertions. Its implementation technique is the focus of the rest of this paper. However, the underlying idea and the general approach is applicable to other DBC notations and tools.

1.4 Approach Overview

To understand JML’s runtime assertion checker [6, 9], imagine runtime assertion checking as a game. The runtime assertion checker only gets to make a move in this game when a subexpression of an assertion exhibits demonic undefinedness. Its move consists of picking a particular value for this demonic undefinedness. The user also gets to make moves; these moves occur whenever a subexpression of an assertion exhibits angelic undefinedness. The user’s moves are unknown to the runtime assertion checker. The runtime assertion checker wins this game by making the whole assertion false, i.e., by reporting an assertion violation; this is considered winning because its goal is to find errors.

The rules for making moves are determined by JML’s semantics. When the runtime assertion checker claims to have won the game by reporting an assertion violation, it must also give reasons (a report to the user) that shows why the assertion is false. One can then check that:

1. the choices made by the runtime assertion checker for each occurrence of demonic undefinedness follow the underspecified total function semantics, and
2. there is no possible choice that could be made for the occurrences of angelic undefinedness that would make the assertion true.

These rules can also be considered to be the soundness conditions for the runtime assertion checker.

A user can win this game by never letting the runtime assertion checker make any choices, which can be done by writing protective assertions [23], and by never making programming or specification mistakes.

JML’s runtime assertion checker tries to win by using a strategy, called a *contextual interpretation*. As assertions are being interpreted, it uses context to track whether an occurrence of demonic undefinedness in a boolean subexpression should be interpreted as true or false, in order to falsify the top-level assertion. To prevent violating the rules when it encounters an angelic undefinedness (e.g., non-executable constructs), it uses the same context in the opposite sense, thus avoiding having a false value for the top-level assertion depend on any occurrence of angelic undefinedness.

In this paper, we explain the contextual interpretation using JML for examples. However, the approach can also be implemented in other assertion languages that use the underspecified total functions semantics for assertions.

The rest of this paper is organized as follows. In Section 2 we describe the undefinedness problem in more detail. In Section 3 we describe the abstract syntax of JML’s specification expressions. The abstract syntax is used in Section 4 to explain the contextual interpretation in detail. We define a set of translation rules from the abstract syntax to assertion checking code, and the translation rules show how undefinedness is interpreted by the runtime assertion checker. In Sections 5 and 6 we discuss various issues and related work, respectively. In Section 7 we conclude this paper with a summary.

2. THE PROBLEM

This section explains the undefinedness problem using JML as an example assertion language and gives more detail about the problem of implementing JML’s semantics in a runtime assertion checker.

Figure 1 shows part of a sample specification written in JML. In JML, specifications are annotated as special comments that start with at-signs (`@`), i.e., `/*@` and `/*@ ... @*/`. The sample specification describes the behavior of the interface `SparseVector` that has a method named `get`. Its behavior is specified using two model fields: `v` and `size`. A *model field* is a specification-purpose field [10]. In this case, a sparse vector is being modeled using a Java array, `v`, and an integer, `size`. As noted by the second invariant, `size` is at least as large as the length of the array `v`.

In JML, a method specification precedes the declaration of the method it specifies. The `requires` clause specifies the pre-condition of the method, and the `ensures` clause

```
public interface SparseVector {
    /*@ model instance double[] v;
    /*@ model instance int size;

    /*@ instance invariant v != null;
    /*@ instance invariant v.length <= size;
    /*@ instance invariant (* size is positive *);

    /*@ requires 0 <= i && i < size;
    @ ensures
    @ ((\result == v[i]) <== (i < v.length))
    @ && ((\result == 0.0) <== (i >= v.length));
    @*/
    double get(int i);

    // ...
}
```

Figure 1: A sample specification in JML.

specifies its post-condition. The pre-condition of the `get` method states that the method should be called with a non-negative argument, `i`, that is strictly less than `size`. The post-condition uses `\result` to describe the return value; it says that the result is the `i`th element of `v` when `i` is strictly less than `v.length`, and `0.0` when `i` is greater than or equal to `v.length`. The post-condition uses JML’s reverse implication operator, `<==`, to express this in a way that closely matches the above English description.

When compiled with the JML compiler [6], the pre- and post-conditions of the `get` method will be checked at runtime just before and right after executing that method’s body, respectively. The question is: what happens if an exception occurs while the compiled runtime assertion checking code is evaluating the pre- or post-conditions? For example, the post-condition has a subexpression `v[i]` that throws an `ArrayIndexOutOfBoundsException` exception if `i` is an invalid index. This leads to the undefinedness problem mentioned above: if `v[i]` throws an exception, what should be the truth value of the post-condition?

Another instance of demonic undefinedness can be seen in the second invariant. It has a subexpression, `v.length`, that would throw an exception if `v` is null. While `v` cannot be null, due to the first invariant, it is not clear from the way these invariants are written, in two separate clauses, which would be evaluated first, hence it is not clear whether the first invariant really protects the second from potential undefinedness. Again, the problem is to assign a definite truth value to the second invariant in the case when `v` is null.

A simple approach, adopted by most DBC tools, of propagating exceptions to the top-level assertion and then either propagating them to the clients or failing the top-level assertion is not adequate. For example, if `v` is null, we would like to say that the first invariant is violated, and not throw an exception. The user has tried to specify an invariant that covers this case, and a normal assertion violation report—indicating the invariant has been violated—should be given, instead of a null pointer exception. Similarly, even if `i` is not less than `v.length` in a call to `get`, this should not cause the evaluation of the post-condition to throw an exception. The user has tried to specify the behavior of `get` using two

cases, and should not be forced to rewrite of the assertion to prevent `v[i]` from being evaluated.

Therefore, an implementation of the underspecified total function semantics is needed. A technique for implementing this semantics should be able to detect as many inconsistencies between the program or specification as possible, as these indicate bugs in either the program or its specification.

Being faithful to the underspecified total function semantics does not necessarily mean being able to catch more bugs. For example, suppose that `SparseVector`'s first invariant (see Figure 1) was omitted (e.g., during early development of the specification). In this version of `SparseVector`, if `v` were null, then evaluating the remaining invariant, `v.length <= size` would lead to a null pointer exception. The semantics allows the runtime assertion checker to substitute an arbitrary value for `v.length`. If it chooses 0 for this value, then the invariant would be satisfied, but then the runtime assertion checker would have lost an opportunity to signal an assertion violation; such an assertion violation would help the user find that the first invariant is needed to be written.

The third invariant in Figure 1 is an instance of angelic undefinedness. For such non-executable constructs it makes little sense to either report an assertion violation or to throw an exception to the user. Users would quickly grow tired of such reports or exceptions, as they are spurious. Instead the runtime assertion checker must implement a semantics that assumes that they do not falsify the overall assertion. To see how subtle this problem is, consider writing the third invariant in the figure as `!(* size is negative *)`; this example shows that one cannot blindly assume that all non-executable boolean expressions are `true`.

Finally, a solution should be efficient in terms of run-time execution costs.

3. JML ASSERTIONS

Assertions in JML are written using a subset of Java expressions, plus a few extensions. Using Java expressions helps make JML a practical tool, as one of the main hurdles to using a new specification language is often lack of familiarity with the notation [11]. All Java expressions may be used in writing JML assertions except for expressions with side-effects, such as assignment expressions and Java's increment and decrement expressions. Another restriction is that only "pure" methods can be called within JML expressions; the purity of an expression is statically checked at compile-time using additional annotations [22]. The extensions to Java's expression syntax include quantifiers, a set comprehension notation, and a few logical operators.

In Section 4 below, we explain our contextual interpretation technique by defining a set of translation rules that map JML expressions into Java program statements. For brevity, we define translation rules only for a subset of JML's specification expressions. The abstract syntax of this subset is shown in Figure 2. We believe that this subset contains enough of the interesting JML expressions to show the essentials of our technique. In addition to the familiar Java expressions, the abstract syntax includes JML-specific logical connectives, such as forward implication (`==>`), and reverse implication (`<==`), universally quantified expressions, and informal descriptions.

The meaning of the Java expressions that are allowed in assertions is largely the same as that of Java expressions, except that JML uses the underspecified total function seman-

Abstract syntax:

```

I ∈ Ident
T ∈ Type
D ∈ Decl
C ∈ Text not containing "*"
E ∈ Expr
E ::= I | E1.I | E0.I(E1, ..., En)
    | !E1 | E1 || E2 | E1 && E2
    | E2 == E1 | E1 != E2
    | E1 ==> E2 | E1 <== E2
    | (\forallall D; E1; E2) | (* C *)
D ::= T I

```

Figure 2: Abstract syntax of a subset of JML's specification expressions.

tics instead of propagating exceptions. The meaning of the implication operators is as in standard (two-valued) logic. The standard meaning for possibly bounded quantifiers is used for the universally quantified expressions. For example, `(\forallall int i; 0 <= i && i < len; a[i] > 0)` is true just when, for all integers `i` that are at least 0 and no more than `len`, `a[i]` is strictly positive. JML does not give a meaning to an informal description, such as `(* s is printed *)`, although they have type `boolean`. Informal descriptions are used as a primary example of non-executable expressions in this paper.

4. CONTEXTUAL INTERPRETATION

We call our strategy for winning the runtime assertion checking game described above a *local, contextual interpretation*. The interpretation is *local* in that undefinedness is interpreted by the smallest boolean expression that encloses the undefinedness. Working on the smallest boolean expression adequately approximates the underspecified total function semantics without much extra overhead or inefficiency. It is *contextual* in that the value for the smallest boolean expression is chosen context-sensitively, depending on the operator involved and the position of the undefinedness occurrence relative to the top-level assertion.

Consider the assertion `!(v.length > size)`. If the variable `v` is null, the evaluation of the subexpression `v.length` completes abruptly by throwing a null pointer exception. The smallest, enclosing boolean expression is `v.length > size`, and it appears inside a negation expression. In such a context, the goal for demonic undefinedness becomes to make the expression true, in order to falsify the top-level assertion. Therefore, if `v` is null, the runtime assertion checker chooses to give the undefined expression `v.length > size` the value true, which makes the entire assertion false.

The notion of contexts plays a crucial role in the interpretation. Without it, the runtime assertion checker might choose false for the subexpression `v.length > size`. However, since the entire assertion negates this subexpression, it would become true with such a choice. So in this example the best strategy is to use the value `true` for the undefined subexpression.

To summarize, the main idea is to use the context of an expression to determine its value upon an occurrence of undefinedness. Each expression has a translation context, which is defined in such a way to follow the semantics.

Table 1: Determining contexts of subexpressions

Enclosing expression, E	Context of $E_1, c(E_1)$	Context of $E_2, c(E_2)$
$!E_1$	$\neg c(E)$	
$E_1 != E_2$	$\neg c(E)$	$\neg c(E)$
$E_1 ==> E_2$	$\neg c(E)$	$c(E)$
$E_1 <== E_2$	$c(E)$	$\neg c(E)$
$(\forall \text{forall } D; E_1; E_2)$	$\neg c(E)$	$c(E)$
all others	$c(E)$	$c(E)$

We define two kinds of contexts: positive and negative. In a *positive context*, an occurrence of an exception is interpreted as **false** by the smallest boolean expression that covers the exception. A *negative context* means that an occurrence of an exception in that context is treated as **true**. That is, demonic undefinedness evaluates to **false** in a positive context, and **true** in a negative context. As a dual of demonic undefinedness, angelic undefinedness evaluates to the opposite value in each kind of context. A top-level assertion is initially interpreted with a positive context. This allows the runtime assertion checker to falsify assertions that, at the top-level, are demonically undefined, and hence to report these as assertion violations. Similarly, if the top-level assertion is angelically undefined, the runtime assertion checker can avoid a false report of a violation by making the assertion true.

When evaluating a boolean expression E , the recursion on E 's subexpressions (E_1, E_2 , etc.) passes along E 's context or its opposite. Following the usual logical notion of positive and negative contexts, the subexpression in a negation expression, both subexpressions in an inequality, the antecedent of either kind of implication expression, and the range expression of a universal quantifier all inherit the opposite of their parent expression's context, but all other subexpressions inherit the contexts of their parent expressions unchanged. Table 1 summarizes the translation contexts for each subexpression. In this table, $c(E)$ denotes the context of overall (parent) expression E , and $\neg c(E)$ denotes the opposite of $c(E)$. For nested contexts, their contexts depend on the level of nested negations. A positive context is nested in an even number negations (possibly zero). A negative context is nested in an odd number of negations.

4.1 Disabling for Equalities

When interpreting equalities and inequalities it is sometimes necessary to disable the contextual interpretation. For example, consider the assertion $(x.f > 0) == (y.g > 0)$. If the variables x and y are both null, then evaluations of subexpressions $x.f$ and $y.g$ both throw a null pointer exception. With the local contextual interpretation, these abrupt completions lead both the left and the right operands of the equality operator evaluate to **false**, as they are the smallest boolean expressions that cover the exceptions thrown. Therefore, the whole predicate evaluates to **true**, as it becomes equivalent to $\text{false} == \text{false}$, and thus no assertion violation is reported by the runtime assertion checker. While the choice of **true** for this assertion's value is consistent with the underspecified total function semantics, we would like to choose **false** and report an assertion violation for it instead.

To be able to report such assertion violations, we disable

the contextual interpretation for subexpressions of equality expressions (and thus also for subexpressions of expressions that desugar into equality expressions, such as inequality expressions). When the contextual interpretation is disabled within an expression, an occurrence of undefinedness in its subexpressions may be propagated to it. However, to obey the rules of logic, this propagation must also be stopped by a subexpression that is decisive. An expression is *decisive* if it can interpret an occurrence of undefinedness as either true or false, under the standard rules of logic. Only a conjunction or disjunction, or expressions that desugar into conjunctions and disjunctions (such as implications) can be decisive. For example, the expression $x.f \ || \ i < 4$ is decisive when the value of i is strictly less than 4, as then an occurrence of undefinedness in $x.f$ does not contribute to the expression's value; instead the evaluation of this example would return **true** even when the contextual interpretation is disabled (see Section 4.3).

4.2 Notation

In the next subsection, we formalize the contextual interpretation by defining a set of translation rules. These translation rules map JML expressions into Java program statements. The rules are presented in a notation shown in Figure 3. The translation function, \mathcal{C} , takes 4 arguments: a JML expression, E , a result variable, r , a context value, p , and a disabled flag, d . Since contexts are either positive or negative, they are represented by simple boolean values; **false** represents a positive context, and **true** a negative context. This particular choice of values was made to ease the implementation. The value that represents the context is used directly for translating demonic undefinedness; to translate angelic undefinedness its negation is used. The disabled flag is **true** when the contextual interpretation is disabled.

Side conditions for rules are written using an underlined sentence preceded by “?”. A rule becomes applicable only when all its side conditions are met. Thus, the first rule in Figure 3 is applied when the contextual interpretation is disabled or when I does not have type boolean; otherwise, the second rule in that figure is applied.

A JML expression is translated into one or more Java statements, usually a try-catch statement or a sequential composition of statements. A JML expression cannot be translated into a Java expression, in general, because Java expressions cannot catch exceptions. For example, as shown in Figure 3, $\mathcal{C}[I, r, p, d]$ denotes a statement that evaluates the variable reference expression I and stores the result into the result variable r . The variable r is assumed to be declared in a surrounding scope, e.g., in the code that incorporates the translated code. The translation uses the boolean value, p , which represents the context, in the second rule of the figure. The first catch block handles angelic undefinedness, and the second handles demonic undefinedness.

4.3 Translation Rules

In this section, we formalize the local contextual interpretation, by defining a set of translation rules for the remaining abstract syntax of JML expressions (see Figure 2).

Figure 4 shows the translation rules for field reference expressions. The first rule in this figure is applied when the contextual interpretation is disabled or when the referenced field is not of boolean type. If the referenced field does not

Translation function:
 $\mathcal{C}: \text{Expr} \times \text{Ident} \times \text{boolean} \times \text{boolean} \rightarrow \text{Stmt}$
 $\mathcal{C}[[I, r, p, d]] \stackrel{\text{def}}{=} \begin{cases} ? \text{ if } d \text{ or if } I\text{'s type is not boolean} \\ r = I; \end{cases}$
 $\mathcal{C}[[I, r, p, d]] \stackrel{\text{def}}{=} \begin{cases} ? \text{ otherwise} \\ \text{try} \{ \\ \quad r = I; \\ \} \text{ catch (JMLNonExecutableException } e) \{ \\ \quad r = !p; \\ \} \text{ catch (Exception } e) \{ \\ \quad r = p; \\ \} \end{cases}$

Figure 3: A sample translation rule.

have type boolean, it cannot be the smallest boolean expression that covers undefinedness, thus the evaluation is not wrapped with contextual interpretation code. As a result, any occurrence of undefinedness is passed to some parent expression. It will be eventually caught by an ancestor of the expression because the top-level assertion is a boolean expression.

The second rule of Figure 4 shows the underlying idea of the contextual interpretation. If the contextual interpretation is not disabled, and if the field I is of type boolean, then the field reference expression $E.I$ is the smallest boolean expression that encloses an occurrence of undefinedness that results directly from the dereference operation, e.g., when E evaluates to null. Thus, the translation rule should handle such an occurrence of undefinedness. For this, the evaluation of the expression $E.I$ is wrapped with contextual interpretation code. It is evaluated inside a `try` block, by using a local variable v . (We assume that such local variables introduced in rules are unique in their scopes.) The expression E is evaluated and its I field is referenced to set the result variable r . If the code inside the `try` block completes abruptly by throwing an exception, then there are two possibilities. If it is caused by demonic undefinedness (e.g., a runtime exception such as `NullPointerException`), the result variable, r , is set to the given context value, p , by the second `catch` clause. If it is caused by angelic undefinedness, the result variable, r , is set to the opposite of the context value, `!p`, by the first `catch` clause. Angelic undefinedness can happen if the expression E is not executable or the field I is a non-executable specification-only field [6]; if evaluated, such an expression will throw an instance of `JMLNonExecutableException`, which indicates an occurrence of angelic undefinedness.

As this is our first rule, it would be instructive to show its application. Consider an assertion `person.isMarried`, where `person` is of a type, say, `Person` that has a boolean field `isMarried`. Assume that the assertion occur at the top-level. We then use $\mathcal{C}[[\text{person.isMarried}, \text{rac}\$r1, \text{false}, \text{false}]]$ to translate the assertion; a non-local variable `rac$r1` receives the result. This translation produces the code shown in Figure 5, using the second rule of Figure 4, since the d argument is `false` and the expression’s static type is boolean, and using the first rule of Figure 3, since the type of `person` is not boolean.

The translation rule for method call expressions, shown in

$$\mathcal{C}[[E.I, r, p, d]] \stackrel{\text{def}}{=} \begin{cases} ? \text{ if } d \text{ or if } I\text{'s type is not boolean} \\ T \ v = \text{null}; \\ \mathcal{C}[[E, v, p, d]] \\ r = v.I; \end{cases}$$

$$\mathcal{C}[[E.I, r, p, d]] \stackrel{\text{def}}{=} \begin{cases} ? \text{ otherwise} \\ \text{try} \{ \\ \quad T \ v = \text{null}; \\ \quad \mathcal{C}[[E, v, p, d]] \\ \quad r = v.I; \\ \} \text{ catch (JMLNonExecutableException } e) \{ \\ \quad r = !p; \\ \} \text{ catch (Exception } e) \{ \\ \quad r = p; \\ \} \end{cases}$$

Figure 4: Translating field reference expressions.

```
try {
    Person rac$v1 = null;
    rac$v1 = person;
    rac$r1 = rac$v1.isMarried;
} catch (JMLNonExecutableException rac$e) {
    rac$r1 = !false;
} catch (Exception rac$e) {
    rac$r1 = false;
}
```

Figure 5: Translation of a top-level assertion `person.isMarried`. The variable `person` is assumed to be of type `Person` and `isMarried` to be a boolean field.

Figure 6, has a similar structure to that of field reference expressions. If the return type is not boolean, one of parent expressions would be such a smallest boolean expression, and thus the expression itself is evaluated without contextual interpretation. However, if the contextual interpretation is not disabled and if the method’s return type is boolean, then the expression is evaluated wrapped with contextual interpretation code. This code is somewhat complicated, because it evaluates multiple subexpressions, some of which might result in angelic undefinedness, and some of which might result in demonic undefinedness. When both sorts of undefinedness might occur, catching the first occurrence of angelic undefinedness and interpreting it with respect to the context would miss opportunities for using potential demonic undefinedness. Since the runtime assertion checker cannot win but only gives up when it encounters angelic undefinedness, it is a better strategy to continue looking for potential demonic undefinedness. That is, demonic undefinedness is given precedence over angelic undefinedness, so that the runtime assertion checker can signal more assertion violations and thus help detect more bugs. For this reason, all subexpressions in a method call are first evaluated, and then the code determines whether any had demonic undefinedness; if not, then angelic undefinedness is tested, and if there was no undefinedness then the method call itself is made in a wrapped context. (In this and all further rules, we assume that local variables declared in a rule are initialized to default values if no initialization is given, but since these

initializations depend on the type of the variable, they are sometimes omitted.)

Figure 7 shows a set of translation rules for logical connectives. The translation rules for implication expressions are defined indirectly by desugaring them into expressions that use only negation, conjunction, and disjunction. The last rule is for translating negation expressions. Because the negation operator changes the translation context, the subexpression, E , is translated in the opposite context, $!p$, and then the result variable, r , is set to the negation of E . The negation expression is a primary example of expressions that change translation contexts.

Figure 8 shows the translation rules for conditional-or ($||$) and the conditional-and ($&&$) expressions. These expressions are short-circuit evaluated. Although not necessary, using short-circuit evaluation saves execution time, because Java programmers tend to write protective expressions such as `x != null && x.f > 0`, where the left-hand subexpression protects the right-hand one from undefinedness. Either the left or the right hand subexpression may yield a decisive value, and when that happens, the decisive value is used without regard to undefinedness of the other subexpression. In JML, therefore, the order of subexpression evaluation does not matter. As with method calls, if both subexpressions are undefined, the translated code favors demonic undefinedness.

4.4 Equality and Inequality

Figure 9 shows translation rules for equality expressions. The first rule is for translating the not-equal-to ($!=$) operator and desugars it into the equal-to ($==$) and the negation operators. The following two rules are for translating the equal-to ($==$) operator. The second of these rules applies whenever the contextual interpretation is not already disabled. It disables the contextual interpretation for its subexpressions. In this case it evaluates both operands and records, in local variables, whether either had angelic or demonic undefinedness. If either had undefinedness, it returns the appropriate value based on its context. Again, demonic undefinedness is given precedence over angelic undefinedness.

4.5 Quantified Expressions

JML provides several forms of quantifiers. A JML quantified expression is either a predicate or a numerical expression. In this subsection, we consider one predicate quantifier, the universal quantifier (`\forall`forall).

In some languages, a quantifier ranges over all existing objects of the quantified types [5]. In Kent and Maung’s extension to Eiffel [17], for example, a type is viewed as a collection of objects, often called a “class extent,” and a quantified variable ranges over all existing instances of its static type. Thus, a quantified predicate is evaluated for each element of the class extent. JML, however, uses the standard logical interpretation of quantifiers and thus considers a quantified variable to range over all *potential* values of the quantified variable that satisfy the range predicate [21, 22]. This affects executability of quantifiers, because if the quantified variable has a reference type, then variable may range over all potential values of that reference type (including null), which is an infinite set. However, in JML one can make such quantifications executable by using a range predicate that requires the quantified variable to be a member of some finite set.

$$\begin{aligned} & \mathcal{C}[[E_0.I(E_1, \dots, E_n), r, p, d]] \\ & \stackrel{\text{def}}{=} \text{? if } d \text{ or if } I\text{'s type is not boolean} \\ & \quad T_0 \ v_0; T_1 \ v_1; \dots T_n \ v_n; \\ & \quad \text{boolean } de = \text{false}, an = \text{false}; \\ & \quad \mathcal{C}_S[[\langle E_0, E_1, \dots, E_n \rangle, \langle v_0, v_1, \dots, v_n \rangle, de, an, p, d]] \\ & \quad \mathcal{C}_U[[de, an]] \\ & \quad r = v_0.I(v_1, \dots, v_n); \\ \\ & \mathcal{C}[[E_0.I(E_1, \dots, E_n), r, p, d]] \stackrel{\text{def}}{=} \text{? otherwise} \\ & \quad T_0 \ v_0; T_1 \ v_1; \dots T_n \ v_n; \\ & \quad \text{boolean } de = \text{false}, an = \text{false}; \\ & \quad \mathcal{C}_S[[\langle E_0, E_1, \dots, E_n \rangle, \langle v_0, v_1, \dots, v_n \rangle, de, an, p, d]] \\ & \quad \text{if } (de) \{ r = p; \} \\ & \quad \text{else if } (an) \{ r = !p; \} \\ & \quad \text{else try } \{ \\ & \quad \quad r = v_0.I(v_1, \dots, v_n) \\ & \quad \} \text{ catch (JMLNonExecutableException } e) \{ \\ & \quad \quad r = !p; \\ & \quad \} \text{ catch (Exception } e) \{ \\ & \quad \quad r = p; \\ & \quad \} \\ \\ & \mathcal{C}_S: \text{seq Expr} \times \text{seq Ident} \times \text{Ident} \times \text{Ident} \\ & \quad \times \text{boolean} \times \text{boolean} \rightarrow \text{Stmt} \\ & \mathcal{C}_S[[\langle E_0, E_1, \dots, E_n \rangle, \langle v_0, v_1, \dots, v_n \rangle, de, an, p, d]] \stackrel{\text{def}}{=} \\ & \quad \mathcal{C}_B[[E_0, v_0, de, an, p, d]] \\ & \quad \text{if } (!de) \{ \mathcal{C}_B[[E_1, v_1, de, an, p, d]] \} \\ & \quad \dots \\ & \quad \text{if } (!de) \{ \mathcal{C}_B[[E_n, v_n, de, an, p, d]] \} \\ \\ & \mathcal{C}_B: \text{Expr} \times \text{Ident} \times \text{Ident} \times \text{Ident} \times \text{boolean} \times \text{boolean} \\ & \quad \rightarrow \text{Stmt} \\ & \mathcal{C}_B[[E, r, de, an, p, d]] \stackrel{\text{def}}{=} \\ & \quad \text{try } \{ \\ & \quad \quad \mathcal{C}[[E, r, p, d]] \\ & \quad \} \text{ catch (JMLNonExecutableException } e) \{ \\ & \quad \quad an = \text{true}; \\ & \quad \} \text{ catch (Exception } e) \{ \\ & \quad \quad de = \text{true}; \\ & \quad \} \\ \\ & \mathcal{C}_U: \text{Ident} \times \text{Ident} \rightarrow \text{Stmt} \\ & \mathcal{C}_U[[de, an]] \stackrel{\text{def}}{=} \\ & \quad \text{if } (de) \{ \text{throw new RuntimeException();} \} \\ & \quad \text{if } (ae) \{ \\ & \quad \quad \text{throw new JMLNonExecutableException();} \\ & \quad \} \end{aligned}$$

Figure 6: Translating method call expressions.

$$\begin{aligned} \mathcal{C}[[E_1 ==> E_2, r, p, d]] & \stackrel{\text{def}}{=} \mathcal{C}[[!E_1 || E_2, r, p, d]] \\ \mathcal{C}[[E_1 <== E_2, r, p, d]] & \stackrel{\text{def}}{=} \mathcal{C}[[E_1 || !E_2, r, p, d]] \\ \mathcal{C}[[!E, r, p, d]] & \stackrel{\text{def}}{=} \mathcal{C}[[E, r, !p, d]] \\ & \quad r = !r; \end{aligned}$$

Figure 7: Translating logical connectives.

```

 $\mathcal{C}[[E_1 \ || \ E_2, r, p, d]] \stackrel{\text{def}}{=}$ 
  boolean  $v = \text{false}$ ,  $de = \text{false}$ ,  $an = \text{false}$ ;
   $\mathcal{C}_B[[E_1, v, de, an, p, d]]$ 
  if (! $v$ ) {
     $\mathcal{C}_B[[E_2, v, de, an, p, d]]$ 
  }
  if (! $v$ ) {  $\mathcal{C}_U[[de, an]]$  }
   $r = v$ ;

 $\mathcal{C}[[E_1 \ \&\& \ E_2, r, p, d]] \stackrel{\text{def}}{=}$ 
  boolean  $v = \text{true}$ ,  $de = \text{false}$ ,  $an = \text{false}$ ;
   $\mathcal{C}_B[[E_1, v, de, an, p, d]]$ 
  if ( $v$ ) {
     $\mathcal{C}_B[[E_2, v, de, an, p, d]]$ 
  }
  if ( $v$ ) {  $\mathcal{C}_U[[de, an]]$  }
   $r = v$ ;

```

Figure 8: Translating conditional expressions.

```

 $\mathcal{C}[[E_1 \ != \ E_2, r, p, d]] \stackrel{\text{def}}{=} \mathcal{C}[[!(E_1 == E_2), r, p, d]]$ 

 $\mathcal{C}[[E_1 == E_2, r, p, d]] \stackrel{\text{def}}{=}$  ? if  $d$ 
   $T_1 \ v_1; T_2 \ v_2;$ 
  boolean  $de = \text{false}$ ,  $an = \text{false}$ ;
   $\mathcal{C}_S[[\langle E_1, E_2 \rangle, \langle v_1, v_2 \rangle, de, an, p, d]]$ 
   $\mathcal{C}_U[[de, an]]$ 
   $r = (v_1 == v_2);$ 

 $\mathcal{C}[[E_1 == E_2, r, p, d]] \stackrel{\text{def}}{=}$  ? otherwise
   $T_1 \ v_1; T_2 \ v_2;$ 
  boolean  $de = \text{false}$ ,  $an = \text{false}$ ;
   $\mathcal{C}_S[[\langle E_1, E_2 \rangle, \langle v_1, v_2 \rangle, de, an, p, \text{true}]]$ 
  if ( $de$ ) {  $r = p$ ; }
  else if ( $an$ ) {  $r = !p$ ; }
  else {  $r = (v_1 == v_2);$  }

```

Figure 9: Translating equality expressions.

In JML, a quantified expression is evaluated by statically identifying predefined patterns that restrict the range of quantification to an interval of values or a finite collection of objects. For a quantification over an integral type, intervals are identified; e.g., $(\forall \text{forall int } i; 0 \leq i \ \&\& \ i \leq 10; a[i] == 0)$ defines an interval between 0 and 10, inclusive. For a quantification over a reference type, collection patterns are identified; e.g., $(\forall \text{forall Student } s; \text{ta.contains}(s) \ || \ \text{ra.contains}(s); s.\text{credits}() \leq 12)$ defines a set consisting of all elements of ta and ra . If such an interval or collection is found at compile time, the quantified expression is executable; otherwise, it is not executable and produces an angelic undefinedness.¹ An executable quantified expression is run by iteratively binding the quantified variable to each element of the interval or collection, and evaluating the predicate for each such binding.

Figure 10 shows one translation rule for the universal quantifier. The rule shown is applicable only when the type of the quantified variable, v , is a reference type. The rule

¹The JML compiler gives warnings when such constructs are not executable.

first calculates a conservative, static approximation of the set of objects, q , that is sufficient to decide the truth of the quantification; if no such set can be found, the quantification becomes non-executable. For each element of q , the desugared predicate $E_1 \implies E_2$ is evaluated. The iteration terminates as soon as the first element that does not satisfy the predicate is found. The evaluation is wrapped in contextual interpretation code because the code calculating the set q may throw an exception. The calculation of the set sufficient to determine the truth of a quantification, is defined by the helper function \mathcal{C}_Q ; details are omitted but can be found in [6]. However, the goal here is to obtain a set q for a quantified expression $(\forall \text{forall } T \ x; E_1; E_2)$ such that the following equivalence holds: $(\forall \text{forall } T \ x; E_1; E_2) \equiv (\forall \text{forall } T \ x; q.\text{contains}(x); E_1 \implies E_2)$. A static analysis is performed on the structure of the range predicate E_1 to find such a set. If the range predicate is not specified, then E_2 's antecedent is analyzed, provided that E_2 is an implication expression [6]. If no such a set is found, the code returned by \mathcal{S}_Q throws an angelic undefinedness so that the quantifier be contextually interpreted by its parent expressions.

```

 $\mathcal{C}[(\forall \text{forall } T \ v; E_1; E_2), r, p, d]]$ 
   $\stackrel{\text{def}}{=}$  ? if not  $d$  and  $T$  is a reference type
  boolean  $b = \text{true}$ ;
  try {
    Collection  $q = \text{null}$ ;
     $\mathcal{C}_Q[[E_1, v, q, p]]$ 
    Iterator  $i = q.\text{iterator}()$ ;
    while ( $b \ \&\& \ i.\text{hasNext}()$ ) {
       $T \ v = (T) \ i.\text{next}()$ ;
       $\mathcal{C}[[E_1 \implies E_2, r, p, d]]$ 
    }
  } catch (JMLNonExecutableException  $e$ ) {
     $b = !p$ ;
  } catch (Exception  $e$ ) {
     $b = p$ ;
  }
   $r = b$ ;

```

$\mathcal{C}_Q: \text{Expr} \times \text{Ident} \times \text{Ident} \times \text{boolean} \rightarrow \text{Stmt}$

$\mathcal{C}_Q[[E, x, r, p]] \stackrel{\text{def}}{=} \dots$

Figure 10: Translating the universal quantifier.

4.6 Non-executable Constructs

A non-executable construct causes an occurrence of angelic undefinedness. The translation of angelic undefinedness is demonstrated by the following rule for the inherently non-executable informal descriptions of JML.

```

 $\mathcal{C}[(\ast \ C \ \ast), r, p, d]] \stackrel{\text{def}}{=}$  ? if  $d$ 
  throw new JMLNonExecutableException();
 $\mathcal{C}[(\ast \ C \ \ast), r, p, d]] \stackrel{\text{def}}{=}$  ? otherwise
   $r = !p$ ;

```

If the contextual interpretation is enabled, the result variable, r , in the translated code is set to the negation of p , thus preventing a false report of an assertion violation. That is, the runtime assertion checker assumes that an informal description always holds.

A similar rule is used for all other non-executable constructs that have type `boolean`, such as quantifiers that are not executable.

For constructs whose results are of non-`boolean` types, the translation rules are defined to throw an angelic undefinedness. For example, the following rule is for translating `\reach` expressions, that denotes the set of all objects “reachable” from the given location [21, Section 3.2].

```
C[\reach(E), r, p, d]  $\stackrel{\text{def}}$ 
  throw new JMLNonExecutableException();
```

A `\reach` expression is translated into a `throw` statement to throw a predefined JML runtime exception that indicates an occurrence of angelic undefinedness. The exception notifies to the parent expressions that a non-executable expression has been encountered. The parent expressions are expected to interpret the exception contextually.

5. DISCUSSION

In JML, one can refer to pre-state expressions in post-state assertions such as post-conditions and history constraints². Referring to pre-state expressions is necessary to specify the behaviors of mutation methods and the properties of mutable objects. JML has two such tools: an old expression and an old variable. An old expression, `\old(e)`, denotes to the value of the expression e in the pre-state. An old variable declaration, `old T x = e`, introduces a specification variable whose value is that of the expression e in the pre-state; an old variable declaration can appear only in a method specification. The basic technique for supporting pre-state expressions is to evaluate them in the pre-state for their potential use in post-state assertions. The results are stored into private fields, and these fields are used to evaluate post-state assertions.

The question here is: how old variables and expressions affect the contextual interpretation? An occurrence of undefinedness in an old expression must be propagated to the post-state expressions that refer the old expression. For this, a special wrapper class is introduced to save the pre-state value into a private field. The wrapper class can encode both demonic and angelic undefinedness, in addition to normal objects and values. If the stored value represents undefinedness, a reference to it (actually a method call) by a post-state assertion throws an appropriate exception, e.g., `JMLNonExecutableException` or `RuntimeException`. Thus, a reference to a pre-state is contextually interpreted by the referring post-state assertion (refer to [6] for details)³.

In Section 4 we defined some of translation rules indirectly by desugaring assertions, e.g., logical connectives (see Figure 7) and equality expressions (see Figure 9). In the actual implementation, however, these expressions are directly translated without being desugared to improve the perfor-

²A history constraint is like an invariant but specifies relationships that should hold for the combination of each visible state and any visible state that occurs later in the program’s execution [24]

³An old variable needs a further treatment as the same variable may be used both in a positive context and a negative context. One solution is to evaluate an old variable’s expression in both contexts, and to use the appropriate value depending on the context of its use. Another possibility is to do contextual interpretation dynamically at runtime by passing the context as an argument. This feature is not yet implemented in JML.

mance of the translated code; the context rules of Table 1 are used for the direct translation. We also note that, because creating an exception is very expensive in terms of runtime speed, our implementation creates only once and reuses exceptions such as `JMLNonExecutableException` and `RuntimeException` to signal an occurrence of undefinedness.

6. RELATED WORK

The simplest approach to undefinedness for runtime assertion checking is to propagate to the user (i.e., out of the runtime assertion checker or directly to a debugger) all exceptions thrown during the evaluation of assertions. This approach is found in the assertion facility (`assert` statements) of the Java programming language [30] and most design-by-contract tools [1, 19, 26]. The main reason that this approach is not used in JML is because one cannot use the standard rules of logic to reason about assertions, as indicated in the introduction. Another reason JML does not adopt this approach is that theorem proving tools, like PVS, use a two-valued logic with underspecified total functions. Hence, to be faithful to JML’s semantics, this approach cannot be used in JML’s runtime assertion checker.

We know of no other DBC tools or runtime assertion checking tools that implement the underspecified total function semantics found in JML.

7. CONCLUSION

We presented an approach that can handle undefinedness occurring during runtime assertion checking. The approach, called a contextual interpretation, interprets an occurrence of undefinedness as either true or false depending on the context. The goal is to preserve the standard rules of logic and to catch as many assertion violations as possible. The contextual approach approximates underspecified total function semantics [13] in a way that is pragmatically useful. It is also able to handle both demonic undefinedness (exceptions) and angelic undefinedness (non-executable constructs). As the approach is not in any way JML-specific, it could also be applied to other formal behavioral interface specification languages and design-by-contract tools. The JML compiler implementing the presented approach is freely available from www.jmlspecs.org with other JML tools and documents.

8. ACKNOWLEDGMENTS

The work of Cheon was supported in part by The University of Texas at El Paso through URI grant 14-5078-6151. The work of Leavens was supported in part by NSF grants CCF-0428078 and CCF-0429567. Thanks to Patrice Chalin and Kui Dai for helpful comments on an earlier draft of this paper.

9. REFERENCES

- [1] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV’01*, 2001. Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001.
- [2] J. A. Bergstra and A. Ponse. Kleene’s three-valued logic and process algebra. *Information Processing Letters*, 67(2):95–103, 1998.

- [3] A. Blikle. Three-valued predicates for software specification and validation. *Fundamenta Informaticae*, XIV:387–410, 1991.
- [4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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.
- [5] R. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Mateo, California, 1994.
- [6] 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. The author’s Ph.D. dissertation.
- [7] Y. Cheon, M. Kim, and A. Perumendla. A complete automation of unit testing for java programs. In H. R. Arabnia and H. Reza, editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP ’05), Volume I, Las Vegas, Nevada, June 27-29, 2005*, pages 290–295. CSREA Press, 2005.
- [8] 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.
- [9] 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, Málaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.
- [10] 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.
- [11] K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, Feb. 1996.
- [12] 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.
- [13] D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, NY, 1995.
- [14] B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Computing Science Institute, University of Nijmegen, 2003.
- [15] C. B. Jones. Partial functions and logics: A warning. *Inf. Process. Lett.*, 54(2):65–67, 1995.
- [16] C. B. Jones and K. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.
- [17] S. Kent and I. Maung. Quantified Assertions in Eiffel. In *Proceedings of TOOLS Pacific 95 (TOOLS 18)*, pages 349–364. Prentice Hall, November 1995.
- [18] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. *Fundamenta Informaticae*, XIV:411–453, 1991.
- [19] R. Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.
- [20] 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.
- [21] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, Department of Computer Science, July 2005. See www.jmlspecs.org.
- [22] 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.
- [23] G. T. Leavens and J. M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10:59–75, 1998.
- [24] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
- [25] B. Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, Oct. 1992.
- [26] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
- [27] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.
- [28] D. S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.
- [29] B. Schieder and M. Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 5(2):73–81, Feb. 1999.
- [30] Sun Microsystems, Inc. A simple assertion facility for the java programming language. Available from <http://java.sun.com/docs/books/jls/assert-spec.html> (Date retrieved: April 2, 2003).