A Runtime Assertion Checker for the Java Modeling Language

Yoonsik Cheon

TR #03-09
April 2003

Keywords: documentation, formal methods, inheritance of specifications, programming by contract, runtime assertion checking, specification language, tools, unit testing, Java language, JML compiler, JML language.

2001 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — modules and interfaces, object-oriented design methods; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, validation, JML; 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, logics of programs, mechanical verification, pre- and postconditions, specification techniques.

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

Contents

List of Figures

Acknowledgements

I thank all those who helped me with various aspects of my research and the writing of this thesis. Gary T. Leavens, my major professor and mentor, guided my graduate study throughout my years at Iowa State. He stimulated my research interest in formal interface specification languages, spent numerous hours with me discussing many ideas and technical details that eventually led to this thesis, and helped me with my writing. I also thank the other members of my thesis committee, Les Miller, Robyn R. Lutz, Don L. Pigozzi, and Clifford Bergman, for their efforts and contributions to this work.

I thank the JML developers and users for helping me develop the JML compiler and writing many excellent bug reports, especially Curtis Clifton, Clyde Ruby, David Cok, Peter Chan, Joseph Kiniry, Marko van Dooren, Roy Patrick Tan, Erik Poll, and Judy Chan. To Curt, I owe a debt of gratitude, for sharing his knowledge on the MultiJava compiler, discussing numerous design and implementation issues, and many hours of pair programming.

I thank all the participants of Spring 2003 Writer's Workshop Seminar at Iowa State, for reading a long introductory chapter of my thesis draft and giving many helpful comments.

I thank the present, and past members of our Programming Languages and Specifications Laboratory (PLSL): Curtis Clifton, Clyde Ruby, Tongjie Chen, George Ushakov, Markus Lumpe, Yaping Jing, Krishna Kishore Dhara, and Tim Wahls. They contributed to my research in one way or another, and made my graduate study at Iowa State a pleasant experience.

Finally, I thank my family — in particular, my parents, for their love and support; my sisters and brothers, for their encouragement and support; Mihee, for always being there for me with love and patience; and Jungwoo, who has been wondering why his daddy always has so much homework to do and no time to play with him.

Abstract

The Java Modeling Language (JML) is a formal behavioral interface specification language (BISL) for Java. JML has many advances including specification-only declarations, specifications of interfaces, stateful interfaces, multiple inheritance of specifications, and behavioral subtyping. An approach to runtime assertion checking of JML assertions is presented and implemented as a JML compiler to translate Java programs annotated with JML specifications into Java bytecode. The compiled bytecode transparently checks JML specifications at runtime. The JML compiler supports separate and modular compilation. The approach brings programming benefits such as debugging and testing to BISLs and also helps programmers to use BISLs in their daily programming.

A set of translation rules are defined from JML expressions into assertion checking code. The translation rules handle various kinds of undefinedness, such as runtime exceptions and non-executable constructs, in such a way as to both satisfy the standard rules of logic and detect as many assertion violations as possible. The rules also support various forms of quantifiers. Specification-only declarations such as model fields, ghost fields, and model methods are translated into access methods; e.g., an access method for a model field is an abstraction function that calculates an abstract value from the program state. The specification state of a stateful interface, due to specification-only fields such as ghost fields, is represented as a separate assertion class. Thus, an object's specification state is distributed over the object itself and one assertion object for each interface that its class implements. Assertion checking is also distributed in that a subtype delegates the responsibility of checking inherited specifications to its supertypes (or the assertion classes of its superinterfaces). The delegation approach supports multiple inheritance, and is modular.

Finally, the effectiveness and practicality of runtime assertion checking is demonstrated by applying it to program testing. An approach is implemented that significantly automates unit testing. The key idea of the approach is to view interface specifications as test oracles and to use the runtime assertion checker as the decision procedure of the test oracles. The approach also shows that the runtime assertion checker can be an effective framework for developing specification-based tools.

Chapter 1

Introduction

In this chapter, I first present an introduction to and the motivation for this dissertation, which is followed by background materials on the Java Modeling Language (JML) and runtime assertion checking. Then, I briefly summarize the problems that are addressed in this dissertation. I also give an overview of my solutions to these problems. Finally, I summarize my contributions. Some parts of this chapter are adapted from my earlier work [28].

1.1 Background

Writing formal interface specifications of program modules such as classes and interfaces can improve the quality of software designs and contribute to the quality of software [110] [148]. This process can help clarify the assumptions that a module makes about its clients and environment; it also helps one identify the module's responsibilities and obligations to its clients [108] [110]. Identifying and precisely specifying responsibilities of modules often lead to a better design that is less coupled and more cohesive. The resulting formal specification is a detailed design document that is abstract, precise, and concise; besides its value during development, such detailed design documentation is especially valuable during the maintenance phase. Some form of specification is also necessary for deciding the success or failure of tests [110].

However, formal interface specifications are seldom used in practice; programmers seldom write formal interface specifications for the program modules they implement [33]. One reason for the average programmer's reluctance to write formal interface specifications is that writing and maintaining such specifications takes a lot of the programmer's precious time, but it does not give an immediate, tangible payoff. The benefits of using formal interface specifications are not obvious to most programmers. Most often such specifications are just another kind of good documents that programmers can refer to and read to understand the modules that they want to use. However, the specification documents must be kept up-to-date as the modules are being changed and refactored, thus adding to the maintenance overhead.

My thesis is that if formal interface specifications bring immediate and tangible benefits in terms of programming activities, such as debugging and unit-testing, that outweigh the burden of writing and maintaining them, programmers would be more willing to write them. The goal of this dissertation is to help programmers reap benefits from specifications as soon as they are written. In particular, the aim is to provide programmers with benefits in debugging and unit testing, which are costly activities that consume much of the time and effort in writing and maintaining software. When this is done, some of the other side benefits of formal specifications, in particular their value as documentation and as an aid in reasoning, will become apparent. In turn this may also help lower costs and improve software quality.

One technique that helps to produce an immediate payoff for writing formal interface specifications is to check specification assertions during the execution of programs. A formal interface specification is just a mathematical formula, but it becomes useful for testing and debugging when it can be executed to check the validity of an implementation. Checking assertions at runtime is a practical and effective means for debugging programs, as Meyer and others have emphasized [110] [134]. It also helps one debug the specifications themselves, and thus improves the quality and accuracy of documentation. Also, checking assertions at runtime can help automate parts of testing [29] [123]. Finally, executing formal specifications is much more practical than using them for formal verification of correctness; the first can be fully automated whereas the latter often requires a user's intervention.

In this dissertation, I address the problems and issues related to the runtime assertion checking of interface specifications. As my research platform, I use the Java Modeling Language (JML), a formal behavioral interface specification language for Java [88] [89] [90]. JML has a syntax that is easily understood by Java programmers, and yet provides many advanced features to facilitate writing abstract, precise, and complete behavioral descriptions of Java classes and interfaces [90].

1.1.1 Runtime Assertion Checking

Assertions are formal facts about the state of a program; they are statements that are true at certain points in program code [62]. Assertions are very useful for both debugging and proving correctness of programs [160]. In order to be able to prove that a program, a method, or a segment of code does what is intended, one needs the ability to make assertions. If an assertion does not hold when the execution control reaches it, one knows that something is wrong with either the program (or the assertion). Therefore, stating what one believes to be true about a program as assertions and checking these assertions at runtime is an effective means of increasing the quality of programs [134].

There may be several ways to support assertions in programming languages, but one of the most popular approaches is to use macro statements that are expanded into appropriate program statements by preprocessors. The main examples are the assertion facilities of C [75] and C++ [46] [145] (e.g., the `assert` macro) and their extensions [36] [104] [134] [158]. Here, assertions are simply boolean statements, embedded into other program statements. If the boolean expressions do not hold at runtime, when the control reaches them, assertion failures are reported. In the Java programming language$^{\mathrm{TM}}$ [55], assertion are provided as built-in statements [147]; however, Java's assertion facility has a similar capability to those of C and C++.

Meyer promoted simple assertions into what is referred to as the *design by contract* (*DBC*) [108] [110] in the programming language Eiffel$^{\mathrm{TM}}$ [109]. Meyer's idea was to use assertions to specify contracts between program modules (i.e., classes) and their clients; contracts are written in Hoare-style pre- and postconditions [62] [63]. The contracts force the clients to fulfill certain conditions each time they use methods provided by the classes; the clients have to check that the *precondition*, stating the requirements of any calls to methods, is satisfied. On the other hand, the implementors have to ensure that each method will have the expected behavior, stated as a *postcondition* expressing the effect of the method. In addition to method pre- and postconditions, class invariants also become a part of contracts. A *class invariant* is a property that must be satisfied by all instances of a class in every stable state. Eiffel's success in DBC contributed to the proliferation of similar notations and tools in many other programming languages, including C [36] [134] [160], C++ [42] [57] [58] [104] [129] [158], Java [5] [44] [47] [71] [81] [82] [116] [117], .NET [3], Python [124] [125] [127], and Smalltalk [20]. These notations and tools vary widely in their techniques and approaches to checking assertions at runtime from simple macro preprocessing and compiling to customized class loaders with the on-the-fly bytecode manipulation.

The notion of DBC rooted in formal methods, but because it uses expressions of the programming language in writing assertions, it is less intimidating (for many programmers) than languages that use lots of special-purpose mathematical notation, like Z [141]. However, standard DBC notations, including that of Eiffel, have several disadvantages. The main disadvantage is the lack of expressiveness, for they syntactically restrict assertions into executable forms, thus many lack quantifiers. Another disadvantage is that they lack support for *abstractions*, the ability to write assertions without using implementation values. It is not possible to write specifications in terms of abstract values as there are no such things as specification-only fields, methods, and types. All assertions are written in terms of program states. Thus, it is often difficult to write abstract, complete specifications in DBC notations. This is troubling because relatively complete specifications are needed if the specifications are to fulfill all their roles (e.g., as test oracles) [29].

While DBC notations emphasize runtime executability of assertions, formal *behavioral interface specification languages* (*BISL*) such as the Larch family of interface languages [26] [27] [59] [86] strive to provide facilities for writing abstract, precise, and complete specifications. Thus, formal BISLs are very expressive with well-defined formal semantics, but unfortunately they lack executability. Formal BISLs are well suited for formally documenting detailed designs of program modules and they are very useful for verifying and reasoning about programs; in fact, complete formal specifications are often a prerequisite for such formal treatment of programs. However, it appears that the lack of executability contributes to the formal BISL's lack of use in practice. The Java Modeling Language tries to bridge the gap between the DBC notations and formal interface specification languages [90].

1.1.2 The Java Modeling Language

The Java Modeling Language (JML) is a formal BISL for Java [88] [89] [90]. JML follows the lead of Eiffel [109] in using the expressions of a programming language to specify the behaviors of program modules. However, JML assertions are not limited to Java expressions; JML adds a rich set of specification constructs including several forms of quantifiers. JML also incorporates many ideas and concepts from the model-oriented approach to specifications [69] such as VDM [66], Z [141], and Larch [59].

In JML, one specifies the behavior of a method in Hoare-style pre- and postconditions [62] [63]. There are also several forms of in-line assertions that can be used as program statements inside method body. In addition to method specifications, one can also state properties about all objects of a particular class or interface, thus indirectly constraining all methods. Two such examples are invariants and history constraints. An *invariant* is a property that must hold in all client-visible states of an object, and a *history constraint* is a relation that should hold between each visible state and all visible states that occur later in the program's execution [101] [103]; a history constraint is used to describe the way that an object changes its state over time.

The following are some highlights of JML that are particularly interesting from the perspective of runtime assertion checking. (In addition to this summary, Chapter 2 provides an overview of JML with enough background materials necessary to read this dissertation.)

- Specification expressions: In addition to Java expressions, JML provides a rich set of specification constructs that can be used to write specification expressions and assertions. This includes logical connectives (`==>`, `<==`, `<==>`, and `<=!=>`), quantifiers (`\forall`, `\exists`, `\min`, `\max`, `\sum`, `\product`, and `\num_of`), a set comprehension notation, an old expression (`\old`) to refer a pre-state value, an object reachability expression (`\reach`), and various other specification constructs. A JML expression must be *pure* in that it should produce no side-effects [48] [90].

- Abstract specifications: A JML specification can be written in terms of abstract (specification-only) values, which, if desired, may be mapped to concrete (program) values. A *model field* provides such a mechanism. A model field is a special field that can only be used inside specifications [95] [96] [139]. By using a model field, one can tune the level of abstractions in one's specifications. There is also a *ghost field* that has no direct correspondence to the program state, but can be used only in specifications. In addition to model and ghost fields, JML also supports specification-only model methods, classes, and interfaces.

- Inheritance, subtyping, and refinement: In JML, a subclass inherits specifications from its direct and indirect superclasses and the interfaces that it directly and indirectly implements [40]. A subinterface inherits specifications from its direct and indirect superinterfaces. The form of specification inheritance ensures the *substitution property* of strong behavioral subtyping [101]. In addition to strong behavioral subtyping, a weak form of behavioral subtyping is also supported, where the subtype's additional methods are relieved from the obligation of obeying the constraints inherited from its supertypes [38] [39] [40]. Another source of specification inheritance is refinement, where a refining type (class or interface) inherits the specifications (including private ones) of its refined type [4]. Inheritance, subtyping, and refinement support reuse, modularization, and maintenance of specifications; e.g., specifications may exist separately from source code.

- Visibility of specifications: JML has a separate visibility control mechanism, in addition to Java's. For example, the same method may have separate method specifications of different visibility, i.e., public, protected, package-visible, and private specifications. The meaning of these visibility levels is the same as in Java. However, a specification of a particular visibility can refer to only those elements that have the same or wider visibility; e.g., a public specification can only refer to public elements, but cannot refer to protected, package-visible, or private elements. Another feature is that the visibility of Java declarations can be widened in JML for specification purposes; e.g., a private field may be declared public for specification purposes.

Other noticeable features of JML include syntactic sugars and specification styles. JML provides various syntactic sugars that ease writing specifications, highlighting various properties for the reader, and making specifications more concise. These include the `non_null` annotation stating that a variable must not be null, the `pure` annotation stating that a method must not have side-effects [15] [96] [97] [115], multiple specification clauses, defaults for omitted specification clauses, specification cases, nested specifications, and redundant specifications [87] [130].

In JML, one can choose different specification styles. A *heavyweight specification* is a "complete" specification with a well-defined meaning for each specification clause; in particular, an omitted specification clause is interpreted as either true or false. By default, a heavyweight method specification is a total correctness specification [41]. In a lightweight specification, an omitted specification clause defaults to `\not_specified`, which means no interpretation is given. Thus, a lightweight specification can be used to state just what is desired by writing individual clauses that one is interested in. By default, a lightweight method specification is a partial correctness specification[1] [41]. An informational description is another mechanism to escape from formality [87].

JML's wealth of features offers many interesting challenges to runtime assertion checking, some of these features will be discussed in Section 1.3.

1.2 Objectives

My ultimate research objective is to help programmers write better programs. I believe that a good way to do this is to bring the benefits of formal interface specifications to programmers in their daily programming work, so that they can leverage the power of formally written interface specifications. My approach for achieving this is to develop practical, effective techniques and tools that programmers can use in their daily programming. The techniques that I envision give an immediate and tangible feedback as soon as one writes formal specifications, without the hassle

[1] It is because an omitted `diverges` clause defaults to false.

often found in heavyweight techniques such as formal verification and reasoning. This will encourage programmers to adopt the techniques widely; programmers are often intimidated by formal methods, perhaps because of lots of special-purpose mathematical notations or the complexity of tools. I envision *runtime assertion checking* as one such technique that can bring immediate and tangible benefits of formal interface specifications. With respect to tools, I envision a compiler that extends the capability of a programming language compiler to generate runtime assertion checking code. The following summarizes my specific research goals.

- To develop a runtime assertion checker for JML. This includes techniques for checking JML specifications at runtime and implementing such techniques into tools, e.g., a runtime assertion checking compiler for Java, called a *JML compiler.*
- To develop a framework for specification-based techniques and tools. A runtime assertion checker can be a basis for developing other specification-based techniques and tools. In particular, I am interested in applying runtime assertion checking to unit testing Java programs.
- To clarify the JML language and its semantics. As a research specification language, JML is evolving with new constructs and features being added, and as thus it currently lacks a fully-formalized semantics. I would like to clarify some of the JML language features and semantics in the process of developing a runtime assertion checker. A runtime assertion checker can be viewed as an operational semantics, to the extent of executable specification constructs.

In sum, the main goal of this dissertation is to create techniques to make formal BISLs executable by checking specifications at runtime, but without losing their expressiveness. The intention is to leverage the power of formal interface specifications by providing immediate and tangible payoffs to programmers. This will contribute for formal BISLs to be practically used in programming and help programmers write better programs.

1.3 The Problem

The main problem is to develop a runtime assertion checker for JML and to show its effectiveness and practicality for use in programming. An essential requirement for runtime assertion checking is transparency; unless an assertion is violated and except for performance measures (time and space), the behavior of the original program should be unchanged[2]. Another requirement is that runtime assertion checking should reflect the semantics of JML. It should be sound in that it does not produce any false positives [48]. The runtime assertion checker should also strive to detect as many potential errors — inconsistencies between specifications and code — as possible; ideally, it should be complete in that it detects all such errors.

However, runtime assertion checking of JML specifications poses several problems. Two problems that need to be mentioned are the lack of a complete formal semantics for JML and the semantic discrepancy between JML and Java. As mentioned earlier, for a runtime assertion checking to be sound, it should be based on the semantics of JML; a runtime assertion checker defines an operational semantics. However, the semantics of JML is not completely formalized and, when my work started, some parts was not well understood yet (e.g., inheritance of specifications and model declarations). Thus, clarifying and resolving any semantic problems should they exist is a prerequisite to runtime assertion checking. In the following, I summarize some of these semantic discrepancies and JML features.

[2] Findler and Felleisen called this property as *coherence* of contract checking [48].

1.3.1 Semantic Discrepancies between JML and Java

Despite the fact that JML is an extension to Java, there are several semantic discrepancies between the JML extensions and Java:

- Stateful interfaces: In Java, interfaces are just declarations; they are *stateless* in the sense that there are no time-varying fields in interfaces. In JML, however, interfaces become *stateful* as one can declare model fields and ghost fields in the interfaces. In the semantics, this means that locations should be allocated somewhere for storing state information attributed to the interfaces.

- Multiple inheritance: Java allows only single inheritance of code while JML supports multiple inheritance of specifications (as specifications are also inherited from interfaces to their implementing classes and extending interfaces). In JML, an interface can have its own fields and (model) methods, and this brings all of the problems associated with multiple inheritance, such as name conflicts [18] [19] [32] [43].

- Refinement: In JML, a type can *refine* another type. The notion of refinement [4] is unique in that everything (including private declarations) is inherited from a refined type by its refining types. There is no corresponding feature in Java.

- Visibility (or accessibility): JML can widen the visibility of a Java declaration. For example, a private field may be declared to be `spec_public`, which means that it has public visibility for specification purposes; the field now becomes accessible to client specifications, but is still not accessible to client code.

1.3.2 Advanced Features of JML

In addition to these semantic discrepancies, JML is an advanced formal BISL with many new features to allow writing abstract, complete behavioral specifications of Java program modules. It offers many interesting challenges to runtime assertion checking. Some of specific problems and features that are addressed in this dissertation include:

- Undefinedness in expressions: The JML expression language is an extension to a side-effect free subset of Java expressions. The use of Java expressions for assertions leads to the problem of potential undefinedness; e.g., a Java expression can throw an exception or an error. Another source of undefinedness is the use of JML-specific specification constructs that are not executable (e.g., informal descriptions). The JML semantics for undefinedness is to substitute an arbitrary expressible value of the correct type for an undefined expression [56] [67] [89, Section 3.1] [93] [94]. The challenge is to make the runtime assertion checker's handling of undefinedness faithful to the semantics of JML, yet at the same time in such a way as to benefit the programmers (e.g., catching more potential errors).

- Quantified expressions: JML has several forms of quantifiers and a set comprehension notation, both of which may not be executable. Often, executability of a quantified expression cannot be determined statically at compile time. Another complication arises if one uses, inside a quantified expression, old expressions (`\old`) to refer to pre-state values, as such a quantification involves values of two different states.

- Inheritance of specifications: In JML, there are three sources of specification inheritance: extension relationships (i.e., subclassing and subinterfacing), implementation relationships (from interfaces to implementing classes), and refinement relationships (from refined types to their refining types). A major concern here is to support separate compilation; e.g., a subclass may be compiled separately from the compilation of its superclasses[3], but the subclass's bytecode should still work even though its superclasses are not compiled with runtime assertion checking code. A main difficulty here is that the inherited specifications may be interpreted in different semantic contexts (or scope) than those of the inheriting specifications. Thus, it is hard to combine inheriting and inherited specifications by desugaring them into one big specification, contrary to an early work in this direction [130].

- Abstract specifications: JML provides facilities such as model and ghost fields, model methods, and model types to write specifications in terms of abstract values, which are not concrete implementation values. Thus, it is necessary to establish a connection between concrete program states and abstract specification states to evaluate assertions involving abstract values.

1.3.3 Research Scope

In this dissertation, I address most of JML's features, but concurrency, subclassing, frame properties, and recently-added features like specification of non-functional properties are left as future research topics. However, it should be mentioned that static analysis techniques have been successfully used to check subclassing contracts [135] and frame properties [22]. It is also interesting to study the usability of tools; however, implementing usability feature needs to be motivated through a new finding or an engineering novelty.

1.4 Approach

My research framework is, for each kind of JML specification clauses and statements, to clarify or formulate its semantics, and then to define a translation rule based on the semantics. The translation rules map JML specification constructs into assertion checking code, and provide a foundation for building the runtime assertion checker. Defining translation rules may necessitate desugaring complex JML specifications into simpler forms and defining detailed structures of assertion checking code. The set of translation rules is viewed as an operational semantics of JML. Once translation rules are defined, the next step is to address the problem of engineering them into Java programs by introducing new techniques and approaches.

In this section, I summarize approaches to the problems and challenges that were identified in the previous section.

1.4.1 What, Where, and When to Check?

The behavior of a Java program module is specified with various specification clauses and statements. Of all these specification constructs, what should be checked at runtime? On one hand, one would like to check as many specification constructs as possible, to detect as many assertion violation errors as possible. This view stems from treating runtime assertion checking as a debugging tool. Debugging is made easier by detecting more potential errors. On the other hand, it may not make a sense to check a certain kind of specifications in a certain circumstance. For example, when a public client calls a public method, checking a private specification may not make a sense because the specification is not visible to the client. The client would be surprised to encounter an assertion violation due to some private specifications unknown to him. This is a DBC aspect of runtime assertion checking; the aim of DBC tools is to detect and blame a contract breach correctly [108] [110] [49]. Ideally, I would like to support both viewpoints in JML. But, if there arises a conflict between two viewpoints, I support the first viewpoint, as my goal is to help programmers write

[3] This requirement may be relaxed in refinement, as the semantics of refinement is to combine all specifications and code of refining and refined types into one type [89].

better programs by providing effective debugging and testing tools. However, such a conflict does not arise often due to modular verification requirements (refer to Section 4.8.1).

The next natural question is where and when to check specification clauses and statements? Debugging is greatly facilitated by isolating and localizing errors. Thus, a module boundary provides a natural place to check specifications; for a specification statement like `assert`, the right place is of course the position where it appears in a block of code. In Java, the basic modularization facility is classes and interfaces, and a module boundary is defined by the set of methods (and fields) that a client can access to use the service provided by a class or an interface. This suggests that the tool should check specifications around method calls.

One can classify various JML specification constructs to help answer the question of when to check specifications. One can identify three different states from the viewpoint of a client who calls a method: pre-states, internal states, and post-states. A *pre-state* is the state where the client calls a method. If a specification concerning the pre-state does not hold, it is in general the client's fault provided that the specification is correct [108]. The client did not call the method in a proper state, and thus the error is in the client code. A *post-state* is a state the method establishes as the result of the call. If a specification concerning the post-state does not hold, it is the implementor's fault [108]. The implementor did not return a promised result or change the state appropriately, thus the error is in the method. An *internal state* is a state between the pre-state and the post-state, hidden from the client. The body of the called method may establish a sequence of internal states, e.g., one for each statement. If anything goes wrong in the internal states (with respect to specifications), then it is the implementor's fault. This classification of program states leads to a similar classification of JML specifications.

- Pre-state specifications: These are specifications constraining the pre-state, such as preconditions, invariants, and `non_null` annotations to both fields and method parameters.

- Post-state specifications: These are specifications constraining the post-state, such as normal and exceptional postconditions, invariants, history constraints, and `non_null` annotations to return objects.

- Internal specifications: These are specifications constraining the internal states. Examples are in-line assertions such as `assert`, `assume`, `unreachable`, and loop invariants and variant functions.

The pre-state specifications are checked in the pre-state, i.e., right after a method call and argument passing but just before the evaluation of the method body. The post-state specifications are checked in the post-state, i.e., right after the evaluation of the method body but just before the method returns (normally or abnormally by throwing an exception). The internal specifications are checked in the internal states, i.e., when the control reaches them during an execution of the code block that contains them.

1.4.2 Assertion Blocks, Methods, and Classes

An important design dimension is to define the structure of runtime assertion checking code and the way it interacts with other assertion checking code and the code being checked. There are several possibilities in organizing assertion checking code:

- Blocks of Java statements: A specification may be translated into a sequence of Java statements, called an *assertion checking block* or *assertion block* for short. The assertion block may be injected into the appropriate position in the method body.

- Separate methods: An assertion checking block may become a separate method of the class being checked, called an *assertion checking method* or an *assertion method* for short. To check an assertion, the assertion method is called in place of the assertion block.

- Separate classes: The assertion methods, instead of being members of the class being checked, may form a separate class, called an *assertion checking class* or an *assertion class* for short. To check an assertion, an assertion object is created and an appropriate assertion method is called on the assertion object.

All three techniques are used in JML. An in-line assertion like a loop invariant is translated into an assertion block; as such an assertion is not inherited, its assertion checking code can be directly injected into the method body (see Section 4.7). Specifications such as pre- and postconditions, invariants, and constraints, are translated into separate assertion methods (see Chapter 4 and 5). The main reason is to facilitate specification inheritance and modularity; to inherit specifications, a subtype can call the corresponding assertion methods of its supertypes. Furthermore, the types can be separately compiled. For a class, the assertion methods become members of the class itself. For an interface, they form a new assertion class for the interface, because all interface methods must be abstract in Java (see Section 5.5). Another role of assertion class is to store the (specification) state of the interface; due to model and ghost fields, an interface becomes stateful (see Section 6.5).

A *wrapper approach* is used to check method specifications. Each method is transformed into a private method, and instead a new *wrapper method* is generated with the same name and signature (see Section 4.3). As a result, all client calls to the original method now go to the wrapper method. The wrapper method is responsible for transparently checking method specifications. For this, the wrapper method delegates client calls to the original method wrapped with appropriate assertion checking. It calls pre-state assertion methods such as preconditions and pre-state invariants before delegating the method call; it calls post-state assertion methods such as postconditions, post-state invariants, and constraints, after delegating the method call. From the client point of view, this has the effect of checking pre-state assertions in the pre-state and post-state assertions in the post-state.

1.4.3 Local Contextual Interpretation

The JML semantics for undefinedness is to substitute an arbitrary expressible value of the correct type for an undefined expression [56] [89, Section 3.1]. My approach is called a *local contextual interpretation*, and the motivation is to detect as many assertion violations as possible while preserving the standard rules of logic. The various causes of undefinedness are classified into two groups: demonic undefinedness and angelic undefinedness. *Demonic undefinedness* is an undefinedness caused by a trouble that should be treated as an assertion violation; examples of the demonic undefinedness are various runtime exceptions and errors. *Angelic undefinedness* is an undefinedness caused by a reason that should not be treated as an assertion violation; a primary example is a non-executable specification construct, such as an informal description.

My approach is to think of runtime assertion checking as a game and apply an optimal strategy for selecting a value for undefinedness (see Section 3.2.3). A strategy would be optimal if it can detect as many assertion violations as possible without false positives. The approach is *local* in that an occurrence of undefinedness is interpreted locally — i.e., only by the smallest boolean expression that encloses the expression that causes the undefinedness. It is *contextual* in that the value of the smallest boolean expression is determined by the expression's context relative to the top-level assertion, such as pre- or postconditions. For demonic undefinedness, the goal is to falsify the top-level assertions under the rules of logic; for angelic undefinedness, the goal is to make them true. For demonic undefinedness, falsifying the top-level assertion allows the runtime assertion checker to signal an assertion violation to the user, and since the rules of logic are always respected, this assertion violation is a real violation that would otherwise go undetected.

1.4.4 Dynamic Delegation

A subtype inherits specifications from its supertypes. My approach for specification inheritance is to delegate the responsibility of checking inherited specifications to the supertypes. The idea is for a subtype's assertion methods, such as precondition methods, to call the corresponding assertion methods of its supertypes (see Chapter 6). The method calls are *delegations* in that, if the supertypes' assertions refer to methods declared by the supertypes but overridden by the subtype, what are actually invoked are the subtype's methods [98] [142]. The delegation approach correctly implements the semantics of JML regarding specification inheritance, as the inherited specifications of supertypes are resolved in the supertypes' environment; e.g., field names are statically resolved and instance method calls are dispatched dynamically. The semantic interpretation is *modular* in that the supertypes' specifications are interpreted in the supertypes themselves, and inherited are interpretations, not specification texts [25]. The delegation approach supports multiple inheritance of specifications; all that are needed is for the subtype's assertion methods to call assertion methods of all its supertypes and combine the results appropriately, e.g., using disjunction for preconditions (see Chapter 6).

Delegation calls are made dynamically by using Java's reflection facility [146]. *Reflection* is the ability of a programming system to make attributes like the invocation, interface, inheritance, and implementation of the objects to be themselves the subject of their own computation [17] [21] [51] [105]. In this dissertation, method calls made by using Java's reflection facility are referred to as *dynamic calls*. The main reason for using dynamic calls is to support *separate compilation*; e.g., a subtype may be compiled separately from the compilation of its supertypes, but the subtype's bytecode should still work even though its supertypes are not compiled with runtime assertion checks. The delegation approach also support *modular compilation* in that compilation of a subtype does not require the source code of its supertypes. In addition, changes in a supertype's specifications do not necessitate recompilation of its subtypes.

1.4.5 Access Methods

Several kinds of access methods are used to support abstract specifications written in terms of specification-only declarations such as model fields, ghost fields, and model methods. An *access method* is a helper method used by assertion checking code to access a specification-only field or method. The underlying idea is, for a specification-only declaration, to generate a unique access method and to translate all reference to the declared field or method into calls to the access method. For example, an access method for a model field calculates (or retrieves) the abstract value of the field from the program state; i.e., it is an abstraction function for the model field [63] [100, pp. 70–71] (see Section 7.2). For a ghost field, a pair of getter and setter methods is generated to read and write to the ghost field (see Section 7.4). For a model method or constructor, an access method forwards all client calls to the private method or constructor that is generated for the model method or constructor; for a constructor, the access method becomes a static factory method (see Section 7.5). Access methods may be called dynamically to support separate and modular compilation.

Access methods are also used to support JML's widening of Java's visibility declarations, e.g., `spec_protected` and `spec_public` (see Section 7.6).

1.5 Implementation

There are several approaches to translating assertions into runtime checking code [126]. The most popular approach is *preprocessing* [5] [81], where assertions are preprocessed to produce instrumented source code that contains both the original source code and runtime assertion checking code. The instrumented code is complied with a programming language compiler to produce executable code. Another approach is a *compilation-based approach*, where assertions are often a built-in programming language feature, and thus they are directly compiled into executable code by a programming language compiler [82] [109] [147]. Yet another approach, for languages based on virtual machines, such as Java, is to manipulate the virtual machine's bytecode to inject assertion checking code. The manipulation can be done either at compile time or loading time, e.g., using a customized class loader

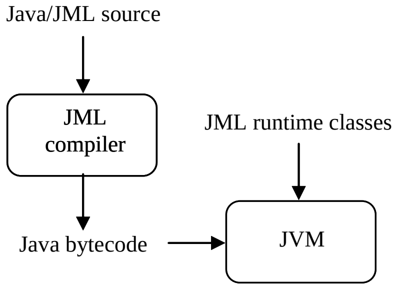

Figure 1.1: Compilation-based approach to runtime assertion checking.

[71] or intercepting the virtual machine's file access [44]. It is also possible to use meta programming for assertion checking, e.g., by introspecting class objects to identify and execute assertion checking code [117].

In JML, the compilation-based approach is adopted, as it is an intuitive and easy-to-use approach (see Figure 1.1). However, contrary to others [82] [147], JML specification constructs are not built-in programming language features, for they annotate Java source code as special forms of comments or exist in separate files. This has an advantage that Java/JML source files can be compiled with a Java compiler like `javac`. The JML compiler compiles Java source programs by translating JML annotations, if any, into runtime assertion checking code. It produces as output Java bytecode (`.class`) files, that can be used in the same way as the output of Java compilers. The bytecode files may run on any Java Virtual Machines (JVMs) except that they may refer to JML-specific runtime classes. In sum, JML compiler is a Java compiler with additional capability of translating JML specifications into automatic runtime checks.

1.5.1 The JML Compiler

My approach for implementing the JML compiler is to reuse the existing source code of JML tools as much as possible, if necessary, by refactoring it. The JML type checker and its underlying Java compiler provide a good code base for the JML compiler. They consist of several compilation passes. My idea is to introduce new compilation passes to generate assertion checking code, and to rewire the whole compilation passes to generate bytecode for both the original and assertion checking code. Ideally, I would like to have a minimal duplication of compilation passes. However, the complexity of assertion checking code and the infrastructure of existing tools make such an optimal solution difficult, and lead to a strategy, called *double-round compilation*.

Figure 1.2 shows the architecture of the JML compiler, `jmlc`, designed as an extension to the JML type checker. The common code base of all JML tools is the MultiJava compiler that extends an open-source Java compiler to support open classes and multiple dispatch [30]. The white ovals represent compilation passes for JML and the gray ovals represent those for MultiJava[4]. The compilation passes for JML extends those of MultiJava; e.g., the parsing pass parses JML specifications, in addition to MultiJava programs. The JML typechecker stops once it completes the typechecking pass.

To implement the double-round compilation strategy, I added a new compilation pass “RAC code generation” after the JML typechecking pass. This pass generates runtime assertion checking code from the typechecked abstract syntax tree, and may mutate the abstract syntax tree to add nodes for the generated checking code. If the added nodes are in the typechecked form, then compilation may proceed directly to the MultiJava's code generation pass; this would be ideal in terms of the compilation speed. However, the complexity of runtime assertion checking code makes it difficult

[4] Several MultiJava-specific compilation passes have been omitted, e.g., the resolution of multiple dispatch methods.

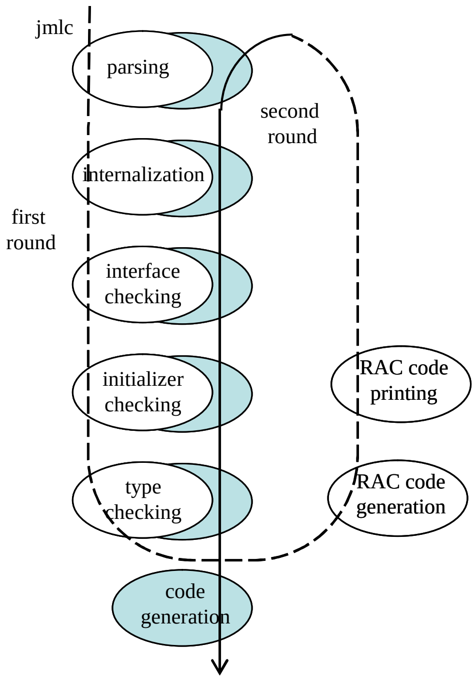

Figure 1.2: The infrastructure of JML compiler (`jmlc`)

to directly build abstract syntax trees in typechecked form. The next optimal solution would be then to proceed to a compilation pass nearer to the code generation pass, e.g., the typechecking pass. But, the existing code base does not allow for an abstract syntax tree to go through the same compilation pass twice, without a major structural surgery to the code. Thus, my approach is to add another new pass "RAC code printing" that writes the new abstract syntax tree to a temporary file, which ends the first round compilation. In the second round, the temporary file is compiled into bytecode by following the MultiJava compilation passes. The reason for the second round following the MultiJava compilation passes is because it is not necessary during the second round to parse JML specifications and they are faster than those of JML passes. The two newly-added compilation passes are about 26K lines of Java code, including program comments.

1.5.2 Reporting Assertion Violations

The JML compiler should translate specifications in such a way that assertion checking code can produce a descriptive message for an assertion violation. There are several ways to report assertion violations, like throwing exceptions, printing error messages, halting the program, and logging. The JML compiler throws exceptions, as this allows a program to automatically cope with assertion violations. The assertion violation exceptions are organized into an exception hierarchy according to the kinds of assertion violations (see Section 4.1.1). This decision also facilitates using the runtime assertion checker as a framework for developing other JML tools; e.g., testing tools may determine test results depending on the kinds of assertion violations encountered during test executions [29].

Another important consideration is the informativeness of assertion violation messages. An assertion violation should be reported in such a way that one can easily identify the source of the error. This is particularly important in JML because an assertion violation can be due to failures of assertions appearing in many different places, sometimes in files that are different from where the code is located. For example, a postcondition violation may be caused by one of several `ensures` clauses or one of many inherited specifications (refer to Chapter 6). In addition to the location, information about the assertion itself, like values of variables and subexpressions involved, would be very helpful to determine the cause of the violation.

1.6 Application — Specifications as Test Oracles

As a case study to demonstrate the practicality and effectiveness of the runtime assertion checker, I have developed an approach that partially automates unit testing of Java classes [29]. The approach frees the programmer from writing unit test code. A successful implementation of the approach also becomes a partial proof of the achievement of my second goal — that the runtime assertion checker can be used as a framework for developing other formal specification-based tools.

The essence of my approach is to view formal specifications as test oracles [2] [123], and to use a runtime assertion checker as the decision procedure for test oracles. The conventional way of implementing test oracles is to compare the test output to some pre-calculated, presumably correct, output. However, my approach is to monitor the specified behavior of the method under test to decide test success or failure. That is, instead of pre-calculated output, the expected behavior is specified in a formal BISL such as JML, and the monitoring is done using its runtime assertion checker. By combining JML with JUnit (a popular unit testing tool for Java [8]), the approach was implemented as a unit testing tool (`jmlunit`), that significantly automates unit testing of Java classes. The tool generates JUnit test classes that rely on the JML runtime assertion checker.

The generated test classes send messages to objects of the Java classes under test. They catch assertion violation errors from test cases to decide if the code failed to meet its specification, and hence that the test failed. If the class under test satisfies its formal specification for some particular test data, no such exceptions will be thrown, and that particular test execution succeeds. Otherwise, an assertion violation will occur, and the test succeeds or fails depending on the type of assertion violation that occurred [29]. If the assertion violation is a precondition violation, then it is further distinguished into two kinds: those that arise from the call to a method and those that arise within the implementation of the method. The first kind of precondition violation, called an *entry precondition violation*, is used to reject test data as not being applicable to the call, while the second, called an *internal precondition violation*, indicates a test failure. When an entry precondition violation occurs, it is not considered to be a test failure, because it indicates that a given test data is outside the domain of the method under test, and thus is inappropriate for test execution. All other assertion violations, including internal precondition violations, are treated as test failures, for they means that the code has failed to meet its specification. In sum, the generated test code serves as a test oracle whose behavior is derived from the specified behavior of the target class. The user is still responsible for generating test data; however the generated test classes make it easy for the user to supply such test data.

It becomes possible to perform unit testing with minimal coding effort and to detect many kinds of errors. Ironically, about half of the test failures were caused by specification errors; this shows that the approach also is useful for debugging specifications. However, the approach requires specifications to be fairly complete descriptions of the desired behavior, for the quality of the generated test oracles depends on the quality of the specifications. Thus, the approach trades the effort one might spend in writing test classes including code to construct expected test outputs for effort spent in writing formal specifications. Formal specifications are more concise and abstract than code, and hence more readable and maintainable. Formal specifications also serve as more readable documentation than testing code, and can be used as input to other tools such as extended static checkers [37]. Other benefits of the approach include improved automation of the testing process, which combines the effect of black-box testing and white-box testing (e.g., through inline assertions), and integrating automated tests with user-written tests (as a programmer may extend and add one's own test methods to the generated test class).

The successful development of the `jmlunit` tool shows that the runtime assertion checker is capable of being a framework for developing other specification-based tools for JML. In fact, the only amendment needed to the runtime assertion checker was making a distinction between entry and internal precondition violations. This distinction is also the main contribution of this case study.

1.7 Contributions

One of the most important contributions of this dissertation is that it demonstrates the possibility of reasonable runtime assertion checking for formal BISLs such as JML. This brings the benefits of runtime assertion checking from simple assertion facilities and design-by-contract tools to full-fledged formal BISLs. This in turn helps such formal BISLs be effectively used by programmers in practice.

The second contribution is that it opens a new possibility in runtime assertion checking by successfully supporting abstract specifications written in terms of specification-purpose fields. It also introduces several new techniques and approaches to engineering runtime assertion checking, including a contextual interpretation of undefinedness, a delegation approach to multiple inheritance of specifications, surrogate classes for (stateful) interface specifications, access methods for abstract specifications, and a pattern-based static analysis to evaluate quantified expressions.

The third contribution is that this dissertation clarifies and resolves many semantic questions about JML, paving a road to a complete formal semantics. The translation rules not only provide a formal basis for runtime assertion checking but also formulate a transformational semantics for JML, and the runtime assertion checker itself is an operational semantics for JML.

The fourth contribution is that this dissertation shows that runtime assertion checking can be an effective framework for developing other specification-based support tools such as unit testing tools. It also provides a valuable framework for JML developers who are interested in developing other JML tools.

Finally, it provides to the Java/JML community a runtime assertion checker, which is a specification-based debugging tool, a specification-based unit testing tool, and a very powerful design by contract tool for Java. The JML compiler and the JML/JUnit tool are available from the JML home page at `http://www.jmlspecs.org`. JML is an open-source project, and thus its source code, including that mentioned in this dissertation, is also available from the JML home page or from the development page at `http://sourceforge.net/projects/jmlspecs`.

1.8 Outline

The rest of this dissertation is structured as follows.

In Chapter 2, I give an overview of the JML language, explaining its important concepts and features. I focus on features that are interesting from the perspective of runtime assertion checking.

In the main body of this dissertation, Chapters 3 through 7, I explain how the JML compiler translates JML specifications into runtime assertion checking code. The general approach is to define rigorous translation rules that map JML specifications into runtime assertion checking code. This involves clarification and formulation of JML semantics, as the translation rules should reflect the semantics of JML. The translation rules show most of important aspects of my approach. Thus, explaining the translation rules constitutes the main body of this dissertation.

In Chapter 3, I discuss the problem of undefinedness and present a unified framework for handling undefinedness in assertions, caused by various reasons such as runtime errors, exceptions, and non-executable constructs. The approach is optimal in that it preserves the semantics of JML with respect to undefinedness, and yet detects as many assertion violations as possible without false positives. In this chapter, I also explain the translation of quantifiers and JML's set comprehension notation.

In Chapter 4, I explain the translation of method specifications such as pre- and postconditions. A method specification is desugared into a simpler form suitable for automatic translation, and desugared top-level assertions such as pre- and postcondition are translated into separate assertion methods such as pre- and postcondition methods. I also describe how the JML compiler translates in-line assertions such as `assert`, `assume`, and loop invariants and variants into assertion checking blocks that are directly injected into the method body.

In Chapter 5, I explain how the JML compiler translates type specifications such as invariants and constraints into runtime assertion checking code. Assertion methods are hosted by the class being checked. However, for an interface, a separate assertion class, called a *surrogate* class, is generated as a static inner class of the interface. The surrogate class is responsible for checking specifications of the interface by hosting all its assertion methods and storing the specification state (e.g., ghost fields) of the interface.

In Chapter 6, I examine the inheritance of specifications in JML, including such features as multiple inheritance and behavioral subtyping. A delegation approach is proposed as a solution; the approach supports all the above features, and separate and modular compilation. The various assertion methods, introduced in the previous chapters, are extended under the delegation approach. In addition, several new assertion methods are introduced to support weak behavioral subtyping. Finally, I explain how the statefulness of interface specifications affects the delegation approach.

In Chapter 7, I consider abstract specifications written in terms of specification-purpose JML declarations such as model fields, ghost fields, and model methods. The goal here is again to support separate and modular compilation, and my approach is to introduce access methods and to use dynamic calls. An access method is generated for each specification-purpose declaration, and each reference to the declared field or method is translated into a dynamic call to the corresponding access method.

In Chapter 8, I demonstrate the practicality and effectiveness of my approach by applying it to specification-based unit testing. The underlying idea is to view JML specifications as test oracles and use the runtime assertion checker as the decision procedure of the oracles.

In Chapter 9, I conclude this dissertation with a summary of my findings, which is followed by an outline of future research directions.

Chapter 2

An Overview of JML

This chapter gives an overview of the Java Modeling Language (JML), introducing its major features. It first shows an example specification, and then explains informally the main features of JML, focusing on features that are interesting from the perspective of runtime assertion checking. For a complete description of JML, one should refer to JML documents such as the reference manual and design documents [88] [89] [90].

2.1 Introduction

JML is a formal BISL for Java to specify the behaviors of and the detailed design decisions on Java program modules such as classes and interfaces [89]. JML combines the practicality of design-by-contract language like Eiffel [109] with the expressiveness and formality of model-oriented specification languages such as VDM [66], Z [141], and Larch [59]. As in Eiffel, JML uses Java's expression syntax in assertions. The advantage of such an approach is that the notation is easier for programmers to learn, and less intimidating than languages that use special-purpose mathematical notations, like Z [141]. However, the approach lacks expressiveness and support for abstractions, because Java's expressions are quite limited compared with specification notations such as higher-order predicate logic formulae. JML remedies this by extending Java's expressions with a rich set of specification constructs, e.g., quantifiers. JML also incorporates many ideas and concepts from model oriented specification languages.

Figure 2.1 shows an example of JML specifications. A JML specification is commonly written as annotation comments in a Java source file[1]. An *annotation comment* starts with `//@` or is enclosed in `/*@` and `@*/`; thus, it is ignored by Java compilers but can be used by JML tools. In JML, the behavior of a method is specified by Hoare-style pre- and postconditions [2] [62]. Preconditions are specified in `requires` clauses, and postconditions are specified in `ensures` clauses. In addition to normal postconditions, JML also supports exceptional postconditions, specified in `signals` clauses. The exceptional postconditions are used to specify what must be true when the method terminates abruptly by throwing an exception. All JML conditions are expressed as boolean expressions in JML's extension to Java's expression syntax; example extensions are `\result` to denote the return value and `\old` to denote pre-state values. The `assignable` clause states a frame property [15] [115]; the method is allowed to assign to locations explicitly listed in the clause and all locations that they depend, but nothing else [95] [96]. In addition to method specifications, properties about all objects of a particular class or interface can be stated, thus indirectly constraining all the methods of the

[1] A JML specification may exist in a separate (specification) file.

[2] It is also possible to describe the behavior of a method by writing abstract code, called a *model program*, in a notation based on the refinement calculus [4]. However, model programs are not addressed in this dissertation.

```
public class BankAccount {
  private /*@ spec_public @*/ long balance;
  //@ public invariant balance >= 0;

  /*@ requires amt > 0 && amt <= balance;
    @ assignable balance;
    @ ensures balance == \old(balance - amt) && \result == balance;
    @ signals (TransactionException) balance == \old(balance);
    @*/
  public long withdraw(long amt) throws TransactionException { /* ... */ }

  // ...
}
```

Figure 2.1: Example JML specification.

class or interface. An example is an invariant, specified by the `invariant` clause, that must hold in all client-visible states.

The following sections summarize the most important features of JML. Section 2.2 explains JML's extension to Java's expression syntax, including several forms of quantifiers and the set comprehension notation. Section 2.3 explains JML's syntax and semantics for specifying methods. Section 2.4 explains type specifications including invariants, constraints, and specification-purpose member declarations. Section 2.5 explains JML's inheritance of specifications. Section 2.6 gives examples for some of JML features explained, and Section 2.7 concludes this chapter, discussing the semantic discrepancies of JML extensions to Java.

2.2 Assertions and Expressions

In JML, specification expressions and assertions are written in Java's expression syntax. However, JML assertions and expressions must be *pure*, meaning that they cannot have any side-effects [90]. Thus, within JML expressions and assertions, one cannot use assignment operators (such as `=`, `+=`, etc.) and increment and decrement operators (`++` and `--`) that would necessarily cause side effects. In addition, only pure methods can be used in JML expressions and assertions. A method is *pure* if it does not have any side-effects to the program state, i.e., does not modify the state. The pureness of expressions, however, does not imply that the expressions always terminate normally. For example, a pure expression `x.length > 0` terminates abruptly by throwing an instance of `NullPointerException` if the variable `x` is `null`.

In addition to side-effect free Java expressions, one can also use JML-specific constructs to write JML expressions and assertions. Table 2.1 summarizes such specification constructs including familiar logical connectives, quantifiers, and other JML-specific constructs. Some constructs such as informal descriptions are inherently non-executable and others such as quantifiers may not be executable. JML supports several kinds of quantifiers: a *universal quantifier* (`\forall`), an *existential quantifier* (`\exists`), *generalized quantifiers* (`\sum`, `\product`, `\min`, and `\max`), and a *numeric quantifier* (`\num_of`). All quantifiers have the same form, `(`Q D`;` E_1`;` E_2`)`, where Q is a quantifier such as `\forall` and `\exists`, D is a variable declaration, E_1 is an optional range predicate, and E_2 is a predicate or an expression. If the range predicate is omitted, it defaults to true. A universally quantified predicate `(\forall` D`;` E_1`;` E_2`)` is equivalent to $(\forall D \bullet E_1 \Rightarrow E_2)$ and an existentially quantified predicate `(\exists` D`;` E_1`;` E_2`)` is equivalent to $(\exists D \bullet E_1 \wedge E_2)$. The quantifiers `\max`, `\min`, `\product`, and `\sum`, are generalized quantifiers that return respectively the maximum, mini-

Table 2.1: JML's extension to Java's expressions. The notation E stands for an expression, T for a type, S for a storage reference expression, I for an identifier, D for a variable declaration, and Q for a quantifier such as `\forall` and `\exists`.

Syntax	Meaning
`\result`	return value
`\old(`E`)`	pre-state value
`\not_modified(`S_1`,...,`S_n`)`	state change
`\fresh(`E_1`,...,`E_n`)`	freshness
`\reach(`E`,`T`,`S`)`	reachable objects
`(* ... *)`	informal description
`\typeof(`E`)`, `\elemtype(`E`)`, `\type(`T`)`	type expressions
`\nonnullelements(`E`)`	non-nullness
`\lockset`	synchronization
`\is_initialized(`E`)`	initialization
`\invariant_for(`E`)`	invariant
T_1 `<:` T_2	subtypeof
`new` T $\{D$ `|` E_1`.has(`I`) &&` $E_2\}$	set comprehension
`(`Q D`;` E_1`;` E_2`)`	quantified expression
E_1 `==>` E_2, E_1 `<==` E_2, E_1 `<==>` E_2, E_1 `<=!=>` E_2	logical connectives

mum, product, and sum of the values of the expression E_2, when the quantified variables satisfy the given range expression E_1 [31]. For example, an expression `(\sum int x; 1 <= x && x <= 5; x)` denotes the sum of values between 1 and 5, inclusive. The numerical quantifier, `\num_of`, returns the number of values for quantified variables for which the range and the body predicate are true.

A set comprehension notation has a similar flavor, and has the form, `new T` $\{D$ `|` E_1`.has(`I`)` `&&` $E_2\}$, where T is a type, D is a variable declaration, I is the declared variable, and E_1 and E_2 are expressions. For example, the following set comprehension expression denotes a `JMLObjectSet` that consists of all `Student` objects found in the set `students` and have the grade `"A"`.

```
new JMLObjectSet {Student s | students.has(s) && s.grade().equals("A")}
```

2.3 Method Specifications

JML provides many extensions and improvements to Hoare-style pre and postconditions, including heavyweight and lightweight specifications, privacy of specifications, normal and exceptional postconditions, frame conditions, and specification redundancy [87] [149] (see Section 2.6 for examples). Various syntactic sugars also facilitate writing method specifications. In addition, JML provides several *in-line assertions*, that can be used in the method body, e.g., `assert`, `assume`, `unreachable`, and loop invariants and variants. In the following, some of these features are briefly described.

2.3.1 Specification Clauses

JML provides several kinds of specification clauses that can be used to specify the behaviors of methods (see Table 2.2). Of theses, the central ingredients are `requires` clauses specifying *preconditions*, `ensures` clauses specifying *normal postconditions*, and `signals` clauses specifying *exceptional postconditions*. Preconditions are predicates that must be satisfied when the specified method is called. Postconditions are predicates that must be satisfied when the method terminates. If the method terminates normally, it must satisfy normal postconditions; otherwise, i.e., if it terminates abruptly

Table 2.2: Method specification clauses.

Clauses	Meaning
`requires`	precondition
`ensures`, `signals`	postconditions
`assignable`	frame condition
`diverges`, `measured_by`	termination
`accessible`, `callable`	subclassing
`when`	synchronization
`duration`, `working_space`	non-functional

by throwing an exception, it must satisfy exceptional postconditions (see Section 2.3.5 for the semantics). If method specifications have multiple specification clauses of the same kind, their predicates are conjoined [130]; e.g., `requires` P`;` `requires` Q`;` is equivalent to `requires` P `&&` Q.

An `assignable` clause specifies a *frame condition* [15] [96] [115]. It states that, from the client's point of view, only the named locations can be assigned during the execution of the method. However, locations that are local to the method (or methods it calls) and locations that are created during the method's execution are not subject to this restriction.

Each method specification clause shown in Table 2.2 has a corresponding clause for specifying a redundant fact. A redundant specification clause is indicated by the suffix `_redundantly` on a keyword, e.g., `requires_redundantly`, and specifies a checkable redundancy. That is, it defines a property that is to be believed to follow from the other properties of the specification [87] [150].

JML also includes several other features that are quite convenient to completely specify the behaviors of methods, e.g., subclassing contracts [135], example specifications [87], model programs [4], synchronization, non-functional properties, and termination. These features are not discussed here, as they are outside the scope of this dissertation.

2.3.2 Heavyweight versus Lightweight

There are two styles one can use to write method specifications in JML: heavyweight specifications and lightweight specifications. A *heavyweight specification*, which starts with one of behavior keywords such as `behavior`, `normal_behavior`, and `exceptional_behavior`, is understood as being "complete" (see Section 2.6 for examples). In a heavyweight specification, therefore, each method specification clause has a well-defined meaning. In particular, an omitted clause is interpreted to be either true or false. A *lightweight* specification is a method specification that does not starts with one of the behavior keywords. In a lightweight specification, the default for an omitted method specification clause is `\not_specified` for every clause, which means no formal interpretation is given. A lightweight specification is intended to state just what is desired by writing individual clauses that one is interested in.

2.3.3 Syntactic Sugars

JML provides various syntactic sugars to write method specifications. Often, specifications written with syntactic sugars are more readable and comprehensible. Two such syntactic sugars that need be mentioned are specification cases and nested specifications. A method specification can consist of several specification cases, combined with the keyword `also` (see Section 2.6 for an example). A *specification case* is a sequence of method specification clauses such as `requires`, `ensures`, and `signals`; it can be lightweight or heavyweight. A method specification written in the case analysis

```
  requires $P_1$;
  assignable $S_1$;
  ensures $Q_1$;
  signals ($E_1$ $e_1$) $R_1$;
also
  requires $P_2$;
  assignable $S_2$;
  ensures $Q_2$;
  signals ($E_2$ $e_2$) $R_2$;
```

```
requires $P_1$ || $P_2$;
assignable $S_1$, $S_2$;
ensures (\old($P_1$) ==> $Q_1$)
  && (\old($P_2$) ==> $Q_2$);
signals ($E_1$ $e_1$) \old($P_1$) ==> $Q_1$;
signals ($E_2$ $e_2$) \old($P_2$) ==> $Q_2$;
```

Figure 2.2: Desugaring specification cases. The left side is desugared to the right side.

```
requires $P_0$;
{|
  requires $P_1$;
  assignable $S_1$;
  ensures $Q_1$;
  signals ($E_1$ $e_1$) $R_1$;
also
  requires $P_2$;
  assignable $S_2$;
  ensures $Q_2$;
  signals ($E_2$ $e_2$) $R_2$;
|}
```

```
  requires $P_0$;
  requires $P_1$;
  assignable $S_1$;
  ensures $Q_1$;
  signals ($E_1$ $e_1$) $R_1$;
also
  requires $P_0$;
  requires $P_2$;
  assignable $S_2$;
  ensures $Q_2$;
  signals ($E_2$ $e_2$) $R_2$;
```

Figure 2.3: Desugaring nested specifications. The left side is desugared to the right side.

style can be desugared into a flat method specification [130]. In Figure 2.2, for example, the method specification in the left-hand-side is desugared into the one in the right-hand-side.

In a specification case, common pre-state method specification clauses such as `requires` can be factored out, giving a *nested method specification*. A nested method specification can also be desugared into a flat method specification [130]. In Figure 2.3, for example, the nested method specification in the left-hand-side is desugared into the one in the right-hand-side. Another specification construct that can be factored out is a declaration of specification variables, which binds a fresh variable for a pre-state expression. Such a specification variable, called an *old variable*, can be used in other specification clauses. For example, the notation `old int s = elems.size();` introduces an old variable `s`. A reference to the variable `s` is semantically equivalent to the expression `\old(elems.size())`.

2.3.4 Privacy of Specifications

As in Java, JML supports the notion of privacy (of specifications), which is explicitly specified in a heavyweight specification and defaults to that of the specified method in a lightweight specification. A specification can be public, protected, package-visible, or private. In JML, however, the same method can have separate specifications of different visibility, i.e., public, protected, package-visible, and private specifications. This allows one to write specifications in a modular, layered fashion, e.g., an abstract specification for clients and a detailed one for implementors. A private specification may record valuable implementation decisions and choices that may be irrelevant to the clients, and such a specification can be changed without affecting the clients.

2.3.5 Semantics

The semantics of a method specification is defined in terms of fully desugared specification, which consists of a specification case (for each visibility) containing only one specification clause for each kind [89] [130]. A method must be called in a state where the method's precondition is satisfied; otherwise, nothing is guaranteed, including even the termination of the call. The state where a method is called is referred to as a *pre-state*. If a method is called in a proper *pre-state*, then there are five possible outcomes of the method execution [89]. The method can (1) return normally, (2) throw an exception, (3) loop forever, (4) abort abruptly, or (5) the JVM may signal an error. If the method terminates normally without throwing an exception, then in that termination state, called a *normal post-state*, the normal postcondition must be satisfied. If the method terminates exceptionally by throwing an exception that does not inherit from the class `java.lang.Error`, then in that termination state, called an *exceptional post-state*, the exceptional postcondition must be satisfied. The method can loop forever or abort (e.g., by calling the method `System.exit`) if the condition specified by the `diverges` clause is met in the pre-state. In JML, one writes a partial correctness specification [62] by writing an explicit `diverges` clause. By default, a heavyweight method specification is a total correctness specification [41] because an omitted `diverges` clause defaults to false. When the precondition is met, in all cases the method is allowed to assign values only to the locations permitted by the `assignable` clause [15] [96] [115]. The method has no obligation if JVM encounters an error, i.e., terminates by throwing an exception that inherits from the class `java.lang.Error` [128].

2.4 Type Specifications

In this dissertation, the term *type specifications* is used to refer to both class specifications and interface specifications. In addition to Java member declarations, type specifications can have JML annotations such as invariants, history constraints, and specification-only member declarations.

2.4.1 Invariants

An *invariant* is a condition that remains true during the execution of a segment of code. In JML, an invariant can be annotated to a type declaration by using the keyword `invariant`, which is called a *type invariant*. A type invariant in general must hold in all client-visible states of all objects of the type. In JML, one can separate invariants that are pertinent only to static fields and methods, called *static invariants*. A static invariant can refer to only static fields and methods; it cannot refer to instance fields and methods. An *instance invariant*, which constrains both static and non-static states of program execution, can refer to both static and instance members. Another important difference between static and instance invariants is that only instance invariants are inherited by subtypes (refer to Chapter 6 for inheritance of specifications).

A type invariant must be preserved by all method of the type, except for helper methods. However, helper methods are exempted from this obligation. A *helper method* is indicated by the JML modifier `helper`, and may be used to establish an internal state that is not visible to the client.

2.4.2 Constraints

A *history constraint*, called a *constraint* for short, is a relation that should hold between each visible state and all visible states that occur later in the program's execution [101] [101]. A constraint is typically used to prescribe the way that values change over time. In JML, therefore, a constraint is typically written with old expressions to relate the current state to the earlier, visible state; a constraint clause starts with the keyword `constraint` (see the example below). An old expression,

`\old(`e`)`, denotes the value of expression e in the pre-state if it appears in method specifications and the value in the earlier, visible state if it appears in constraints.

As with invariants, JML makes a distinction between *instance constraints* and *static constraints* [89]. An instance constraint should be respected by only instance methods, whereas a static constraint should be respected by both instance and static methods. A method *respects* a history constraint if the post-state of the method invocation is in the relation specified by the constraint. Thus, a constraint can be thought of as being implicitly conjoined to the postcondition of a method. Since there is no well-defined pre-state for constructors, and no post-state for finalizers, an instance constraint does not apply to constructors and finalizers. A helper method is also exempted from respecting history constraints, just like it does not have to preserve invariants.

It is also possible to specify constraints that are applicable only to a specific set of methods. For example, the following constraint, that may appear in the specification of the class `BankAccount` (see Section 2.1), constrains only the method `withdraw`; however, if the `for` clause is omitted, it becomes a universal constraint that constrains all methods.

```
//@ public constraint balance <= \old(balance) for withdraw();
```

2.4.3 Abstract Specifications

It is important for specifications to abstract away from implementation details. JML provides *model* and *ghost fields* to form the abstract values of Java data types [89] [95] (see Section 2.6 for an example). They are specification-only fields, and their declarations are the same as those of Java except that they have the JML modifiers `ghost` and `model`. They can be used only in specifications and do not have to appear in the implementation. Their values are abstract in the sense that one is not concerned with their time or space efficiency [63]. The difference between model and ghost fields is that a model field's value is implicitly given as a mapping from the program state whereas a ghost field's is directly manipulated with specification statements. A `represents` clause defines this mapping for a model field by stating how the field is related to program fields (see Section 2.6 for an example). That is, the `represents` clause specifies an *abstraction function* [63] [100, pp. 70–71] or *relation* [6] [45] [92] [138] [140] for the model field. The `set` specification statement that is similar to Java's assignment statements, is used to set the value of a ghost field (refer to Section 4.7.3 for the `set` statement).

JML also support specification-purpose methods, called *model methods.* It is not necessary for a model method to have a body. A model method is often used to define an abstraction function or relation for a model field.

In JML, there are also model classes or interfaces, which are specification-purpose types.

2.5 Inheritance of Specifications

In JML, a subclass inherits specifications from its superclasses and interfaces that it implements. An interface also inherits specifications of the interfaces that it extends. What kinds of specifications are inherited? All specifications of appropriate visibility levels (i.e., public and protected) are inherited, including method specifications, invariants, constraints, specification-only declarations, `represents` clauses, `depends` clauses, and so on. However, only instance specifications are inherited; e.g., static invariants are not inherited. The semantics of specification inheritance reflects that of code inheritance in Java. A program variable appearing in assertions is statically resolved, and an instance method call is dynamically dispatched.

An important feature of JML's specification inheritance is that its semantics supports a behavioral notion of subtyping [40] [101]. The essence of behavioral subtyping is summarized by Liskov and Wing's *substitution property*, which states that a subtype object can be used in place of a supertype's object [101] [102] [103]. In JML, specifications are inherited in such a way to ensure the substitution property [40]. Preconditions are disjoined, and postconditions are conjoined in the form of $\bigwedge(\texttt{\textbackslash old(}p_i\texttt{)} \Rightarrow q_j)$, where p_i is a precondition and q_i is the corresponding postcondition. Type assertions such as invariants and constraints are conjoined.

JML is distinguished in its support for a weak form of behavioral subtyping. A subtype has to preserve the history constraints specified by its supertypes. In particular, the subtype's *additional methods* — new methods of the subtype that do not override inherited ones — have to establish in the post-state the instance constraints of its supertypes. This stringent requirement is called a *strong behavioral subtyping.* If the subtype's additional methods are relieved from obeying the type constraints inherited from its supertypes, it is called *weak behavioral subtyping* [38] [39] [40]; however, in the weak form, the subtype's overriding methods still have to preserve the inherited type constraints. In JML, weak behavioral subtyping is specified by using a `weakly` annotation; if the `weakly` annotation is omitted, it defaults to the strong behavioral subtyping. For example, the following specification says that the class `S` is a strong subtype of the class `T`, but a weak subtype of the interface `I`.

```
public class S extends T implements I  /*@ weakly @*/ { /* ... */ }
```

A refinement relationship, specified by the `refine` declaration, is another source of specification inheritance in JML [89]. A refining class or interface inherits all the specifications of the refined class or interface; in particular, private specifications are also inherited. A refinement relationship is defined for the same type, whose code and specifications may exist in separate files. A refined specification may be more abstract or describe a partial behavior, whereas a refining specification may be more detailed or complete. Another possibility is that a refined type contains only specifications while a refining type contains only code. Thus, refinement also help to modularize specifications.

2.6 An Example

This section presents a JML specification that shows some of the JML features explained in the previous sections. Figure 2.4 shows a JML specification for the class `StackAsArray`. The first line is a JML annotation that imports the class `JMLObjectSequence` from the package `org.jmlspecs.models`. The keyword `model` highlights that this is a specification-purpose import; thus it does not have to appear in an implementation. The class `JMLObjectSequence` implements a mathematical model for sequences of objects, i.e., immutable sequences [90], and is used to model stacks as sequences.

A stack is represented as an array of objects by the field named `array`. The `non_null` annotation states that the field `array` cannot be null; it is equivalent to writing an invariant clause, `private invariant array != null;`. The field `index` represents the current top of the stack and its value is increased as the stack grows. The `invariant` clause states that the value of the field `index` is restricted to a certain range that is a proper index to the array `array`. The keyword `private` indicates that this invariant is for the implementor of the class; such a specification is not visible to clients.

A stack is modeled as a sequence of objects by the model field named `elems`. The model field `elems` is visible to clients, as it is declared `public`. The idea is that even if stacks are implemented as arrays, they can be viewed by clients as sequences. The `represents` clause defines a correspondence between the abstract state, represented by `elems`, and the program state, represented by program fields such as `array` and `index`. The value of the model field `elems` is equal to an expression `JMLObjectSequence.convertFrom(array, index + 1)`. The `depends` clause states that the program fields `array` and `index` can be modified by methods that are allowed to modify the model field `elems` [95] [96]. As both clauses are private, they are not visible to clients, thus hiding the representation.

The specification of the constructor shows a use of specification cases. It consists of a public specification case and a private specification case, separated by the keyword `also`. The first specification

```
//@ model import org.jmlspecs.models.JMLObjectSequence;
public class StackAsArray {
  private /*@ non_null @*/ Object[] array;

  private int index = -1;
  //@ private invariant -1 <= index && index < array.length;

  /*@ public model JMLObjectSequence elems;
    @ private represents elems <-
    @   JMLObjectSequence.convertFrom(array, index + 1);
    @ private depends elems <- array, index; @*/

  /*@ public normal_behavior
    @  requires size >= 0;
    @  assignable elems;
    @  ensures elems.isEmpty();
    @ also
    @ private normal_behavior
    @  requires size >= 0;
    @  assignable elems, array, index;
    @  ensures array.length == size && index == -1; @*/
  public StackAsArray(int size) { /* ... */ }

  /*@ public normal_behavior
    @  assignable elems;
    @  ensures elems.equals(\old(elems.insertBack(e)));
    @  ensures_redundantly elems.length() == \old(elems.length() + 1); @*/
  public void push(/*@ non_null @*/ Object e) { /* ... */ }

  /*@ public behavior
    @  assignable \nothing;
    @  ensures !isEmpty() && \result == elems.last();
    @  signals (Exception e) isEmpty()
    @    && (e instanceof IllegalStateException); @*/
  public /*@ non_null @*/ Object top() { /* ... */ }

  //@ ensures \result == elems.isEmpty();
  public /*@ pure @*/ boolean isEmpty() { /* ... */ }

  // ...
}
```

Figure 2.4: Specification of the class `StackAsArray`.

case is visible to clients (indicated by the keyword `public`), while the second is for implementors of the class (indicated by the keyword `private`). The public specification case is written in terms of the public model field `elems`, and the private specification is written in terms of the private program fields `array` and `index`. (In JML, a specification of a particular visibility can refer to only those elements that have the same or wider visibility; e.g., a public specification can refer to public elements, but not protected, package-visible, or private elements.) The keyword `normal_behavior` indicates that the method cannot terminate abruptly by throwing an exception, thus an exceptional postcondition, specified by the `signals` clause, cannot appear in these specification cases. The specification states that clients should instantiate a `Stack` object with a non-negative size, and the implementors must ensure that such an object establish a state corresponding to an empty stack by indirectly modifying the model field `elems`, i.e., assigning to the program fields `array` and `index`.

The precondition of the method `push` is omitted, and thus defaults to true. The `non_null` annotation to the parameter `e` means that clients cannot call this method with a null object. The `non_null` annotation in this context is equivalent to conjoining a predicate `e != null` to each precondition (including the default one, if omitted); thus the effective precondition of the method `push` becomes `e != null`. The postcondition states that pushing an item onto a stack is equivalent to adding the item at the end of the sequence represented by the model field `elems`. The method specification also shows an example of redundant specification clauses, i.e., `ensures_redundantly`.

The specification of the method `top` is interesting in that the method can terminate either normally or abruptly by throwing an exception `IllegalStateException`; this fact is highlighted by the keyword `behavior` (instead of `normal_behavior` or `exceptional_behavior`), which also suggests that the user should specify both normal and exceptional postconditions. The `assignable` clause states that the method `top` is an observer in that it is not allowed to change any object. This frame axiom is necessary because, if omitted, in such a specification case, it defaults to `assignable \everything`, stating that it can mutate any reachable objects. The normal postcondition says that if the stack is not empty, the return value must be the last element of the sequence `elems`. The exceptional postcondition states that if the stack is empty, the method can throw an `IllegalStateException`, but no other kinds of exceptions; technically, the `signals` clause states that if the specified method terminates abruptly by throwing the declared exception (in this case, `Exception`), the given predicate must be satisfied in that exceptional post-state. The method declaration has a `non_null` annotation, which in this case means that, if the method terminates normally, the return value must not be null. Thus, it is equivalent to conjoining a predicate `\result != null` to the normal postcondition.

Finally, the method `isEmpty` is specified in the lightweight style, and thus the omitted precondition defaults to `\not_specified`. The visibility keyword is also omitted; the specification becomes publicly visible, as the visibility defaults to that of the specified method in lightweight specifications. The specification also shows another kind of annotations, a `pure` annotation. A `pure` annotation states that the specified method has no side-effects; it is desugared into an `assignable` clause of the form, `assignable \nothing;`, which states that the method is not allowed to change any non-local locations.

2.7 Discussion

JML is an extension to Java in that it adds a set of specification facilities to Java. However, the extension introduces some semantic discrepancies between Java and its JML extension. The first such discrepancy is due to model declarations. A model or ghost field can be declared in an interface. This allows an interface to be *stateful* in the sense that the interface can now have state information. This contradicts Java, where an interface is stateless in that it cannot have an instance field. A model method introduces a similar mismatch between JML and Java, for an interface model method can have a body.

To be able to write specifications for interfaces is very important, for interfaces provide a logical boundary between clients and implementors of program modules, and thus are ideal places to attach contracts. However, an interface specification brings an unpleasant consequence that Java avoided by not allowing code in the interface. That is, Java permits only single inheritance of code, from superclasses to subclasses, but JML allows multiple inheritance of specifications; specifications are also inherited from interfaces to implementing classes and subinterfaces. This complicates the semantics of JML, raising the problem of name conflicts due to multiple inheritance. As field names are statically resolved, a textual copy semantics, which textually copies down inherited specifications to the inheriting type, does not provide a suitable semantic framework.

JML's introduction of refinement is unique in that it has no corresponding feature in Java or other specification languages. A refining type inherits everything from a refined type including even its private specifications and code. The refinement relationship is restricted to single inheritance; a refining type can have at most one direct refined type.

In JML, encapsulation of specifications is another important feature. A specification is declared public, protected, package-visible, or private with the same meaning as in Java. An important aspect of this is that a specification of a particular visibility cannot refer to entities of a narrower visibility; this prevents a specification of a wider visibility (e.g., public) from exposing an entity of a narrower visibility (e.g., private). However, a specification is often written in terms of the internal representation (i.e., private fields). Thus, JML allows one to widen the visibility of a Java declaration. A private or package-visible field, method, or type can become protected or public for specification purposes. This means that such a field, method, or type now can be used in the specifications of subclasses and client programs. However, note that it still cannot be used in the code of subclasses and client programs. In sum, a declaration can have multiple scopes of visibility, one for JML specifications and the other for Java code.

Chapter 3

Expressions and Assertions

This chapter discusses how the JML compiler evaluates JML expressions and assertions. The approach is explained by defining translation rules from JML expressions into runtime assertion checking code. Two particular aspects of the translation are examined in detail, the undefinedness problem and the translation of quantifiers.

3.1 Introduction

JML extends Java's expression syntax to write expressions and assertions (see Section 2.2). JML's extension includes quantifiers, a set comprehension notation, and other specification constructs. All Java expressions can be used in writing JML expressions except for expressions like assignments, increment operators, and decrement operators, that may cause side-effects. Another restriction is that only pure methods can be called within JML expressions; the pureness of an expression is statically checked at compile-time. Thus, except for JML-specific constructs and forbidden Java expressions, JML expressions look the same as Java expressions. Indeed, the meaning of JML expressions is for the most part the same as that of Java expressions. However, there are several important semantic differences between JML and Java expressions.

- Abrupt completion: In Java, the evaluation of an expression may *complete abruptly* by throwing an exception [55, Section 15.6]. No interpretation is given to the value of an abruptly-completed expression, except that the abrupt completion always has an associated reason (i.e., `throw`). In JML, however, the semantics is to substitute an arbitrary expressible value of the correct type for an expression that throws an exception [90].

- Order of evaluation: In Java, some operators are order-sensitive while they are not in JML. For example, conditional operators such as `&&` and `||` evaluate the right-hand operand only if the value of the left-hand operands is not conclusive; they are short-circuit evaluated. If the whole expression completes normally, the order does not matter in terms of the result. Otherwise, however, it can make a difference between Java and JML. Consider an expression `x.length > 0 || true`, where `x` is an array. In Java, the result can be either `true` or a throw of `NullPointerException` (which happens if `x` is null). In JML, the result is always `true`. If `x` is null, the term `x.length` takes an arbitrary value of type `int`, and thus the subexpression `x.length > 0` becomes either `true` or `false`. However, the subexpression does not contribute to the result, because `true` is a zero-element of disjunction (`||`). In short, JML expressions obey the standard rules of logic even in the presence of undefinedness.

- Executability: In JML, not all expressions are executable. Some like informal descriptions are inherently non-executable, and others like quantifiers may not be executable.

A formal specification is written abstractly without worrying about computational concerns such as exceptions, runtime errors, and executability. However, the runtime assertion checker has to address these concerns to evaluate assertions. The semantics of JML is to substitute an arbitrary expressible value of the correct type for undefinedness. In this chapter, a unified framework is presented to handle undefinedness in assertions, caused by various reasons. The idea is to categorize undefinedness into demonic and angelic undefinedness, and to apply a (game) strategy to select the optimal value based on the category. A chosen strategy is optimal if it obeys the formal semantics and can detect more errors than any other strategy. For demonic undefinedness (e.g., runtime exceptions), the goal is to falsify the top-level assertion (e.g., pre- or postconditions) under the rules of logic, and for angelic undefinedness (e.g., non-executable constructs), the goal is to make it true. The approach taken is optimal, because it preserves the semantics of JML with respect to undefinedness, and can detect as many assertion violations as possible without false positives.

JML has several forms of quantifiers and a set comprehension notation, both of which may not be executable. An extensible framework is provided to host multiple techniques to evaluate them. However, the primary evaluation strategy is to statically analyze a quantified expression and to determine a set of objects or values that is sufficient to decide the result of the expression. If such a set can be identified, the quantified expression is translated into code that evaluates the expression part of the quantifier iteratively with the quantified variable bound to each element of the set. Otherwise, the quantified expression is translated into (angelic) undefinedness so that the expression's context determines an optimal value.

The rest of this chapter are organized as follows. Section 3.1.1 introduces notations that are used throughout this chapter. Section 3.2 explains the undefinedness problem, and introduces an approach to the undefinedness problem, called a *local, contextual interpretation*. The approach is explained in detail by defining a set of translation rules from JML expressions to assertion checking code. Section 3.3 explains how quantified expressions are evaluated by the runtime assertion checker. Section 3.4 describes related work. Section 3.5 discuss an anomaly of the contextual interpretation, and Section 3.6 summarizes this chapter.

3.1.1 Notations

In this chapter, I explain my approaches by defining a set of translation rules. The translation rules map JML expressions into Java program statements. However, to reduce the complexity of presentation, translation rules are defined only for a subset of JML expressions that are enough to show the essentials of the approaches. Figure 3.1 shows the abstract syntax of JML expressions to lead my discussion in this chapter. In addition to the familiar Java expressions, the abstract syntax includes JML-specific logical connectives such as equivalence (`<==>`), inequivalence (`<=!=>`), forward implication (`==>`), and reverse implication (`<==`). In Section 3.3, I will add quantifiers and the set comprehension notation to the abstract syntax.

Figure 3.1 also shows a sample translation rule. A JML expression is translated into Java code fragment, a sequence of Java statements. The reason for this is that JML expressions (e.g., quantifiers) are often too complex to be translated directly into Java expressions, and need a special handling to cope with undefinedness. The translated code in general becomes the body of assertion checking methods such as pre- and postcondition checking methods (see Chapter 4). The translation function, $\mathcal{C}$: Expression × Identifier × boolean → Program, maps JML expressions to Java program code. For example, $\mathcal{C}[\![E_1 \texttt{>} E_2, r, p]\!]$ denotes a piece of Java program code that evaluates the comparison expression E_1 > E_2 and stores the result into the program variable r. The translated code may use the boolean value, p, as the default value to cope with undefinedness such as exceptions and non-executable constructs (see Section 3.2 for details). The variable r is assumed to be declared in the outer scope, e.g., by code that incorporate the translated code. The translated code may introduce local variables (e.g., v_1, v_2, and e in this example), and such variables are assumed to be unique in their scopes.

```
Abstract syntax:
 $I \in$ Identifier
 $E \in$ Expression
 $E$ ::= $E_1$ <==> $E_2$ | $E_1$ <=!=> $E_2$
       | $E_1$ ==> $E_2$ | $E_1$ <== $E_2$
       | !$E$
       | $E_1$ || $E_2$ | $E_1$ && $E_2$
       | $E_2$ == $E_2$ | $E_1$ != $E_2$
       | $E$.$I$
       | $E_0$.$I$($E_1$,...,$E_n$)
       | ...

Translation function:
$\mathcal{C}$: Expression × Identifier × boolean → Program
$\mathcal{C}[\![E_1$ > $E_2,\ r,\ p]\!] \overset{\text{def}}{=}$
  try {
    int $v_1$ = 0;
    int $v_2$ = 0;
    $\mathcal{C}[\![E_1,\ v_1,\ p]\!]$
    $\mathcal{C}[\![E_2,\ v_2,\ p]\!]$
    $r$ = $v_1$ > $v_2$
  } catch (JMLAngelicException $e$) {
    $r$ = !$p$;
  } catch (Exception $e$) {
    $r$ = $p$;
  }
```

Figure 3.1: Abstract syntax of JML expressions and sample translation rule.

3.2 The Undefinedness Problem

Assertions in specifications are written abstractly in mathematical terms and formula, and their meanings are well defined in terms of pure mathematical models. The mathematical models are not concerned with computational problems such as exceptions, runtime errors and executability of assertions. However, to evaluate the assertions at runtime, the runtime assertion checker has to handle these concerns in the computation model. The problem becomes more acute for JML, as it uses Java expressions for writing assertions. A specification assertion can throw an exception or raises a runtime error because it is an expression of the underlying programming language. In JML, another source of undefinedness is non-executable specification constructs such as informal descriptions. In both mathematical and computational models, the semantics must be precise about the meanings of assertions upon an occurrence of undefinedness; otherwise, the semantics would be incomplete and leave a possibility of different and potentially conflicting interpretations.

3.2.1 An Example

Figure 3.2 shows the specification of the method `push` of the interface `StackType`. The behavior of the method is specified by using a model field named `contents` (see Section 7.2 for model fields). The method specification has both pre- and postconditions. The runtime assertion checker evaluates the pre- and postconditions to determine a potential assertion violation. But, what if an exception occurs while evaluating the pre- or postcondition? For example, the postcondition has a term

```
//@ model import org.jmlspecs.models.JMLObjectSequence;
public interface StackType {
 /*@ model instance non_null JMLObjectSequence contents
   @   initially contents.isEmpty(); @*/

 /*@ assignable contents;
   @ ensures contents.size() == \old(contents.size() + 1) &&
   @   contents.get(0) == elem &&
   @   (\forall int i; 0 < i && i < contents.size();
   @      contents.get(i) == \old(contents.get(i-1))); @*/
 void push(/*@ non_null @*/ Object elem);

  // ...
}
```

Figure 3.2: JML specification with potential undefinedness.

`contents.size()`, that may lead to an abrupt completion of the evaluation. If the field `contents` is null, the expression throws a `NullPointerException`. It is also possible that the method `size` of the class `JMLObjectSequence` may throw some runtime exception. If either of these happens, what should be the values of the expression `contents.size()` and enclosing, parent expressions up to the postcondition? Does the postcondition hold?

A practical approach is needed to evaluate assertions in the presence of undefinedness. The approach should be faithful to the semantics of JML, e.g., preserving the standard rules of logic, and also be able to detect as many errors as possible so that the runtime assertion checker can be used as an effective tool for debugging and testing. Being faithful to JML's semantics does not necessarily means catching more errors. Consider, for example, the method `push`'s postcondition in the interface `StackType` (see Figure 3.2). The evaluation of the term `contents.size()` may lead to an abrupt completion, e.g., with a reason `NullPointerException`. Then, the semantics of JML allows the runtime assertion checker to substitute an arbitrary value for `contents.size()`. The checker may choose a value that satisfies all enclosing expressions, including the postcondition itself. Such an interpretation is legitimate, but by doing so the the checker would loose an important opportunity of catching a potential error.

Other requirements includes (1) to provide a unified framework to handle various causes of undefinedness, such as exceptions, runtime errors, and non-executable constructs, (2) to be definite about top-level assertions such as preconditions, postconditions, and invariants, and (3) to be efficient in terms of implementation. The reason for the second requirement is to let the runtime assertion checker determine assertion violations and, if necessary, take appropriate actions. This is particularly important to tools that rely on the runtime assertion checker, e.g, testing tools to decide test successes or failures (see Chapter 8 for such an application).

3.2.2 Demonic versus Angelic

Undefinedness can have various causes, such as exceptions, runtime errors, and non-executable constructs. I make a distinction among various causes of undefinedness, from the perspective of runtime assertion checking. The distinction help to provide a unified framework to cope with undefinedness. A cause is *demonic* if it should be treated as an error by the runtime assertion checker. Examples of demonic causes include various kinds of exceptions and runtime errors. A cause is *angelic* if it should not be treated as an error by the runtime assertion checker. A primary example of angelic causes is a non-executable specification construct such as informal descriptions. This is because the runtime assertion checker cannot prove it is false. The accurate value is not known in the computation model — i.e., it is not computable due to various reasons, but mostly because the expression is not amenable to a practical execution scheme. Therefore, it would be unsound to assume that such an assertion does not hold; it may produce a false positive.

I use the terms *demonic* and *angelic undefinedness* to refer to undefinedness due to demonic and angelic causes respectively. In the next section, I explain how this distinction is used to cope with undefinedness in a unified framework.

3.2.3 Contextual Interpretation

I call my approach a *local, contextual interpretation.* The key idea is to think of runtime assertion checking as a game and apply an optimal strategy for selecting a value for undefinedness. For demonic undefinedness, the goal is to falsify the top-level assertions, such as pre- or postconditions, under the rules of logic; for angelic undefinedness, the goal is to make them true. The interpretation is *local* in that undefinedness is interpreted by the smallest boolean expression that encloses the undefinedness. 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 occurrence relative to the top-level assertion.

As an example, consider the following `requires` clause, where `x` is an array variable.

```
//@ requires !(x.length <= 0);
```

If the variable `x` is null, the evaluation of the term `x.length` completes abruptly by throwing a `NullPointerException`. The smallest, enclosing boolean expression is `x.length <= 0`, and it appears inside a negation expression. In such a context, the goal for demonic undefinedness like `NullPointerException` becomes to make it true, so that it has an opportunity to falsify the top-level assertion, i.e., the precondition. Therefore, the expression `x.length <= 0` evaluates to true, and thus the precondition becomes false.

The notion of contexts plays a crucial role in the interpretation. Without it, one might choose false for the expression `x.length <= 0` in the hope of falsifying the top-level assertion, i.e., the precondition. However, the precondition would become true with such a choice of value. Thus, the main idea here 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 ensure the rules of logic. An expression's context is determined based on the operator involved, and also relative to the context of its parent expression.

I define two kinds of contexts: a negative context and a positive context. In a *negative context*, an occurrence of an exception is interpreted as `false` by the smallest boolean expression that covers the exception. A *positive context* means that an occurrence of an exception in that context is treated as `true`. In sum, demonic undefinedness evaluates to `false` in a negative context, and true in a positive context. As a dual of demonic undefinedness, angelic undefinedness evaluates to the opposite value in each kind of contexts. A top-level assertion, such as pre- and postconditions and invariants, takes a negative context, and a subexpression either inherits the context of its parent expression or takes the opposite context. In general, negation-style expressions take the opposite context, and all other expressions inherit the contexts of their parent expressions. The reason that top-level assertions start with negative contexts is that the runtime assertion checker should be able to treat an occurrence of an exception at the top level assertion as an assertion violation.

Table 3.1 shows how the translation contexts are determined. The contexts of expressions are recursively defined by the structures of the expressions and with respect to the contexts of the parent expressions. Let $\epsilon(E)$ denote the context of expression E. It is either `true` or `false`; `true` for the positive context, and `false` for the negative context. The contexts of E's subexpressions are defined in terms of $\epsilon(E)$. As shown, only the negation operator, forward and backward implication operators, and the universal quantifier change contexts of their subexpressions.

Table 3.1: Translation contexts for expressions

Structure of enclosing expression, E	Context of subexpression E_1, $\epsilon(E_1)$	Context of subexpression E_2, $\epsilon(E_2)$
!E_1	!$\epsilon(E)$	N/A
E_1 || E_2	$\epsilon(E)$	$\epsilon(E)$
E_1 && E_2	$\epsilon(E)$	$\epsilon(E)$
E_1 ==> E_2	!$\epsilon(E)$	$\epsilon(E)$
E_1 <== E_2	$\epsilon(E)$	!$\epsilon(E)$
E_1 <==> E_2	$\epsilon(E)$	$\epsilon(E)$
E_1 <=!=> E_2	$\epsilon(E)$	$\epsilon(E)$
(\forall D; E_1; E_2)	!$\epsilon(E)$	$\epsilon(E)$
(\exists D; E_1; E_2)	$\epsilon(E)$	$\epsilon(E)$

```
$\mathcal{C}[\![E.I, r, p]\!] \overset{\text{def}}{=}$ ? if $I$'s type is boolean
  try {
    $T$ $v$;
    $\mathcal{C}[\![E, v, p]\!]$
    $r$ = $v$.I;
  } catch (JMLAngelicException $e$) {
    $r$ = !$p$;
  } catch (Exception $e$) {
    $r$ = $p$;
  }

$\mathcal{C}[\![E.I, r, p]\!] \overset{\text{def}}{=}$ ? otherwise
  $T$ $v$;
  $\mathcal{C}[\![E, v, p]\!]$
  $r$ = $v$.I;
```

Figure 3.3: Translating field reference expressions.

3.2.4 Translation Rules

In this section, I formalize the local contextual interpretation, by defining a set of translation rules from the abstract syntax of JML expressions (see Figure 3.1) to Java program code. The translation function, $\mathcal{C}$: Expression × Identifier × boolean → Program, takes three arguments, a JML expression, a result variable, and a context value (see Section 3.1.1). The third argument, the context value, specifies the translation context which is either true or false. The context value true means that the translation should be done in a positive context, and false means that the translation should be done in a negative context. The context value is for translating demonic undefinedness; thus, angelic undefinedness should be translated in the opposite context.

Figure 3.3 shows the translation rules for field reference expressions. A special notation, an underlined sentence preceded by "?", is used to specify side conditions. A rule becomes applicable only when all its side conditions are met. Thus, the first rule is applied when the field is boolean type; otherwise, the second rule is applied.

The first rule is interesting, as it shows the underlying idea of the local contextual interpretation. If the field I is of type boolean, then the field reference expression $E.I$ becomes the smallest boolean expression for any occurrence of undefinedness in the expression, e.g., when E turns out to be 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. (All such local variables are assumed to be initialized with their default values, but these details are suppressed in the translation rules.) 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 (e.g., a non-executable construct), the result variable, r, is set to the opposite of the context value, $!p$, by the first `catch` clause. This happens, for example, if the referenced field I is a non-executable model field (see Section 7.2); a reference to such a model field throws an instance of `JMLNonExecutableException` which is a subclass of the class `JMLAngelicException`. The exception class `JMLAngelicException` indicates an occurrence of angelic undefinedness.

As this is the first rule for contextual interpretation, it would be instructive to show an example. Consider a `requires` clause, `//@ requires person.isMarried;`, where `person` is of type, say, `Person` which has a boolean field named `isMarried`. Then the precondition `person.isMarried` is translated into the following code.

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

As the expression `person.isMarried` appears in the top-level assertion (i.e., the precondition), it is translated in the negative context, i.e., with the context value false. With this translation, if the variable `person` becomes null, the result variable `rac$r1` is set to false, i.e., the precondition becomes violated. If the expression appears in a positive context, e.g., `!person.isMarried`, then upon an occurrence of undefinedness the result variable is set to the opposites to the values shown in the above code. Thus, a null value for the variable `person` still leads to a precondition violation.

The second rule of Figure 3.3 is applied when the referenced field is not of type boolean. As its type is not boolean, the field reference expression cannot be the smallest boolean expression that covers undefinedness, thus the evaluation is not wrapped with contextual interpretation code. As a result, any undefinedness, if occurring, would be passed up to its parent expression. Such an occurrence of undefinedness would be eventually be interpreted by some ancestor expression, as the top-level assertion (e.g., the precondition) is a boolean expression.

The translation rules for method call expressions, shown in Figure 3.4, has the structure very similar to those for field reference expressions. If the method's return type is `boolean`, then the expression itself is the smallest boolean expression that covers any occurrences of undefinedness inside the method body. Thus, as in the field reference expressions, the expression is evaluated wrapped with contextual interpretation code. Otherwise, one of parent expressions would be such a smallest boolean expression, and thus the expression itself is evaluated without contextual interpretation. However, its subexpressions such as the receiver and arguments are still evaluated with contextual interpretation. A helper function, $\mathcal{C}_M$, is used to factor out the common translation, i.e., the translation of the method call expression itself without contextual interpretation.

Figure 3.5 shows a set of translation rules for logical connectives. The translation rules for equivalence and implication expressions are defined indirectly by desugaring them into those that

$$\mathcal{C}[\![E_0\texttt{.}I\texttt{(}E_1,\ldots,E_n\texttt{)},r,p]\!] \stackrel{\text{def}}{=} \underline{\text{? if } I\text{'s return type is boolean}}$$

```
try {
  $\mathcal{C}[\![\texttt{M}]\!]E_0\texttt{.}I\texttt{(}E_1,\ldots,E_n\texttt{)},r,p$
} catch (JMLAngelicException $e$) {
  $r$ = !$p$;
} catch (Exception $e$) {
  $r$ = $p$;
}
```

$$\mathcal{C}[\![E_0\texttt{.}I\texttt{(}E_1,\ldots,E_n\texttt{)},r,p]\!] \stackrel{\text{def}}{=} \underline{\text{? otherwise}}$$

$$\quad \mathcal{C}_M[\![E_0\texttt{.}I\texttt{(}E_1,\ldots,E_n\texttt{)},r,p]\!]$$

$\mathcal{C}_M$: Expression × Identifier × boolean → Program

```
$\mathcal{C}_M[\![E_0\texttt{.}I\texttt{(}E_1,\ldots,E_n\texttt{)},r,p]\!] \stackrel{\text{def}}{=}$
    $T_0$ $v_0$; $T_1$ $v_1$; $\cdots$; $T_n$ $v_n$;
    $\mathcal{C}[\![E_0,v_0,p]\!]$ $\mathcal{C}[\![E_1,v_1,p]\!]$ $\cdots$ $\mathcal{C}[\![E_n,v_n,p]\!]$
    $r$ = $v_0$($v_1$,...,$v_n$);
```

Figure 3.4: Translating method call expressions.

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

Figure 3.5: Translating equivalence, implication, and negation expressions.

use only negation, conjunction and disjunction[1]. The translation rule for negation expressions is interesting. Because the negation operator changes the translation context, the subexpression, E, is translated in the opposite context, i.e., !p. The result variable, r, is set to the negation of the expression E. The negation expression is a primary example of expressions that change translation contexts.

Figure 3.6 shows the translation rules for conditional expressions. As in Java, both the conditional-or (||) and the conditional-and (&&) expressions are short-circuit evaluated; the left-hand subexpression is evaluated first, and then the right-hand subexpression is evaluated, but only if the left-hand subexpression cannot determine the value of the whole expression. However, there is an important difference here between Java and JML. Even if the evaluation of the left-hand subexpression completes abruptly, e.g., by throwing an exception, the evaluation of the whole expression proceeds normally, i.e., by evaluating the right-hand subexpression, if necessary. Such an abrupt completion is interpreted by the subexpression itself as either true or false (see the translation rules for equality expressions and method call expressions later in this section). In JML, therefore, the order of evaluating subexpressions does not matter.

Figure 3.7 shows translation rules for equality expressions. The first rule is for the operator

[1] In the JML compiler, however, they may be translated directly without desugaring for the performance of the translated code. For this, the context rules shown in Table 3.1 should be used.

```
$\mathcal{C}[\![E_1$ || $E_2,\ r,\ p]\!] \overset{\text{def}}{=}$
  $\mathcal{C}[\![E_1,\ r,\ p]\!]$
  if (!r) {
    $\mathcal{C}[\![E_2,\ r,\ p]\!]$
  }
```

```
$\mathcal{C}[\![E_1$ && $E_2,\ r,\ p]\!] \overset{\text{def}}{=}$
  $\mathcal{C}[\![E_1,\ r,\ p]\!]$
  if (r) {
    $\mathcal{C}[\![E_2,\ r,\ p]\!]$
  }
```

Figure 3.6: Translating conditional expressions.

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

$\mathcal{C}[\![E_1$ == $E_2, r, p]\!] \overset{\text{def}}{=}$ $\underline{\text{? if } E_1\text{'s type is boolean}}$
  $\mathcal{C}_E[\![E_1$ == $E_2, r, p]\!]$

$\mathcal{C}[\![E_1$ == $E_2, r, p]\!] \overset{\text{def}}{=}$ $\underline{\text{? otherwise}}$
  try {
    $\mathcal{C}_E[\![E_1$ == $E_2, r, p]\!]$
  } catch (JMLAngelicException $e$) {
    $r$ = !$p$;
  } catch (Exception $e$) {
    $r$ = $p$;
  }

$\mathcal{C}_E$: Expression × Identifier × boolean → Program
$\mathcal{C}_E[\![E_1$ == $E_2, r, p]\!] \overset{\text{def}}{=}$
  $T_1$ $v_1$;
  $T_2$ $v_2$;
  $\mathcal{C}[\![E_1,\ v_1,\ p]\!]$
  $\mathcal{C}[\![E_2,\ v_2,\ p]\!]$
  $r$ = $v_1$ == $v_2$;
```

Figure 3.7: Translating equality expressions.

not-equal-to (!=), which is desugared into the equal-to (==) and the negation operators. The second and third rules, for the operator equal-to (==), is a bit involved. The complication is again due to a possibility of equality expressions being the smallest boolean expression that encloses an occurrence of undefinedness. Such a possibility arises only when the subexpressions are not boolean expressions; if they are boolean, they or their subexpressions will be the smallest such boolean expressions. If there is such a possibility, then the evaluation is wrapped with contextual interpretation code, i.e., try-catch statement. Otherwise, it is evaluated without the contextual interpretation. A new translation function $\mathcal{C}_E$ is a helper function that factors out common translations.

3.2.5 Non-executable Constructs

The translation rules in the previous section can handle angelic undefinedness such as non-executable specification constructs. How does such undefinedness occur?

A non-executable construct causes an occurrence of angelic undefinedness. For example, the translation rules for boolean-typed JML specification constructs such as informal descriptions [89,

```
I ∈ Identifier
T ∈ Type
D ∈ Declaration
E ∈ Expression
E ::= ···
    | (\forall D; E; E)
    | (\exists D; E; E)
    | (\sum D; E; E)
    | (\product D; E; E)
    | (\min D; E; E)
    | (\max D; E; E)
    | (\num_of D; E; E)
    | new T {D | E.contains(I) && E}
D ::= T I
```

Figure 3.8: Extended abstract syntax of JML expressions

Section 3.1], are defined as follows.

$$\mathcal{C}[\![\texttt{(* ... *)}, r, p]\!] \stackrel{\text{def}}{=} r\ \texttt{=}\ \texttt{!}p\texttt{;}$$

Because the context value, p, is for demonic undefinedness such as exceptions, the result variable, r, in the translated code is set to the negation of p, thus implementing an optimistic view on the non-executable constructs. The interpretation means that an informal description always holds.

For specification constructs whose results are of non-`boolean` types, the translation rules are defined to throw angelic undefinedness. For example, the following rule is for translating `\reach` expressions, that denotes the set of all objects "reachable" from the given location [89, Section 3.2].

$$\mathcal{C}[\![\texttt{\textbackslash reach(}E\texttt{)}, r, p]\!] \stackrel{\text{def}}{=}$$

```
if (true) {
  throw new JMLNonExecutableException("\reach(E)");
}
```

A `\reach` expression is translated into a `throw` statement that signals a predefined JML runtime exception, `JMLNonExecutableException`. The class `JMLNonExecutableException` is a subclass of `JMLAngelicException`. The exception notifies to the parent expressions that a non-executable expression has been encountered. The parent expressions will interpret the exception contextually (see translation rules in the previous section).

3.3 Quantified Expressions

3.3.1 Abstract Syntax Extended

JML provides several forms of quantifiers such as `\forall`, `\exists`, `\max`, `\min`, `\product`, `\sum`, and `\num_of` [31], and the set comprehension notation, `new T {}` [89, Section 3.1]. (see Section 2.2). A quantified expression is either a predicate or a numerical expression, and a set comprehension expression denotes a set of objects, e.g., an instance of the class `JMLObjectSet`.

Figure 3.8 shows the abstract syntax of JML expressions extended with quantifiers and the set comprehension notation. The abstract syntax is simplified; e.g., the syntax allows only one variable in a quantified variable declaration (D), whereas one can declare more than one variable in JML.

3.3.2 Semantic Clarification

There are several subtle questions that need clarification for the correct evaluation of JML quantifiers by the runtime assertion checker. What is the range of a quantified variable if it is not constrained by the range predicate at all? Does the range include only the currently existing objects, all the objects that have existed in the past, or even all the objects that may exist in the future? Is the value `null` included too? For value types (i.e., Java primitive types) like `int`, the answer is clear because they are intrinsically immutable and lack identities. When quantifying over such types, we are interested in ranging over values regardless of their memory locations. Indeed, the Java's equality (`==`) operator tests for value equality, not reference equality, for primitive types. For reference types such as classes, interfaces and array types in Java, there are several possible interpretations.

- All existing objects versus all possible objects.
- Including the value `null` versus excluding it.

In object-oriented database query languages, a quantifier ranges over all currently existing objects and the value `null` is an invalid element [23]. Kent and Maung took a similar position for Eiffel [74]. The idea here is to view a type as a collection of its instances, often called a *class extent* or *type extent*. Thus, a quantified variable ranges over all existing instances of its (declared) type, and the assertion is evaluated for each element of the type extent. This approach, which is called a *type extent-based approach* in this dissertation, is practical and effective in a system where type extents are explicitly maintained by the runtime system. However, I doubt its applicability to an object-oriented programming language such as Java. The meaning of "existing" is not clear and precise. Included are all objects regardless of their accessibility or reachability? Some objects may not be visible to a particular client, or others may be lost (e.g., no reference at all). Performing a garbage collection may change the meaning of a predicate. Worse, Java has special *reference objects*, that are garbage collected at the discretion of the Java Virtual Machine (JVM), e.g., when the memory is low [146].

The JML document states that a quantifier ranges over all potential values of the variable declared which satisfy the range predicate [89, Section 3.1]. It further states that when a variable declared is a reference type, it may be null or may reference to an object not constructed by the program. The variable may reference to an object of the subtype of its declared type; however, this is not explicitly stated in the JML document, Thus, JML takes a position that the universe of quantification consists of all possible objects (of correct type) including the value `null`. This semantics of JML makes the type extent-based approach to be unfaithful to the semantics of JML, though it may provide a good approximation. It is unfaithful in that the approach is not sound; there is a possibility of false positives.

3.3.3 Pattern-based Static Analysis

In the literature, several techniques and methods have been proposed to make quantified assertions executable for different formal specification notations and languages (see Section 3.4). However, there is no single, accepted, and universal approach that caters to all needs, and it is unlikely that there would be one in the future. It would be also true in JML, especially considering various forms of quantifiers. Thus, my approach for the JML compiler is to provide an extensible framework to host multiple evaluation strategies; this will facilitate introducing new evaluation techniques and strategies in the future. The JML compiler tries a chain of evaluation strategies until it comes up with a strategy that can handle a given quantified expression or all strategies are attempted at which point the expression becomes non-executable. However, the current, primary evaluation strategy is based on the static analysis of patterns, which is discussed in this section.

The *pattern-based static analysis approach* identifies predefined patterns to restrict the range of a quantifier to a finite collection or interval. If such a collection or interval is found, the quantified variable is bound to each element of the collection or interval, and the expression of the quantifier is evaluated iteratively. Otherwise, the expression becomes angelic undefinedness, i.e., translated into code that throws a `JMLNonExecutableException`, thereby it is contextually interpreted by parent expressions. For a quantification over an integral type such as `int`, patterns for intervals are identified; e.g., the quantified expression `(\sum int x; x > 1 && x < 5; x)` defines an interval between 1 and 5, and thus can be evaluated by the runtime assertion checker. For a quantification over a reference type, patterns for collections are identified; e.g., the quantified expression `(\forall Student p; ta.contains(p) || ra.contains(p); p.credits() <= 12))` defines a set consisting of elements of both `ta` and `ra`, and thus becomes executable.

Figure 3.9 shows the translation rules for the universal quantifier. The given rules are applicable only when the type of the quantified variable, v, is a reference type. The rule first calculates a conservative, static approximation of a set of objects, q, that is sufficient to decide the truth of the quantification. If no such a set can be calculated, the quantification is regarded as being non-executable. For each element of the set q, the desugared expression E_1 `==>` E_2 is evaluated; note that the iteration terminates as soon as the first element that does not satisfy the predicate is found. The evaluation is wrapped with contextual interpretation code because the code calculating the set q may throw an exception; for non-boolean-typed quantifiers like `\sum`, the quantifiers become the smallest boolean expression that covers undefinedness that may occur in E_2.

The calculation of a set that is sufficient to determine the truth of a quantification, is defined by the helper functions $\mathcal{C}_Q$ and $\mathcal{Q}$. The goal here is to obtain a set q for a quantified expression `(\forall` T x`;` E_1`;` E_2`)` such that the following equivalence holds.

$$(\texttt{\textbackslash forall}\, T\ x\texttt{;}\ E_1\texttt{;}\ E_2) \ \equiv\ (\texttt{\textbackslash forall}\, T\ x\texttt{;}\ \texttt{.contains(}x\texttt{);}\ E_1\, \texttt{==>}\, E_2)$$

Ideally, this set would be as small as possible. The idea is to find such a set through a simple, static analysis on the structure of the range predicate E_1. If the range predicate is not specified, then E_2's antecedent is analyzed provided that E_2 is an implication expression; however, these details are not shown in the translation rules. The helper function $\mathcal{Q}$ calculates such a set, and it uses a simple pattern and the structures of expressions. The definition of $\mathcal{Q}$ uses several set notations like $\cup$ and $\cap$ to denote the result of calculation. The notation $\top$ denotes the universe, i.e., the set of all possible objects, and thus it satisfies such properties as $\top \cup s = \top$ and $\top \cap s = s$. The helper function $\mathcal{C}_Q$ translates the calculated set into the corresponding Java code; the details are suppressed here, but it is a direct translation of the set union and intersection operators. One thing to note here is the treatment of non-executable quantifiers. If the result of $\mathcal{Q}$ is the universe, $\top$, then it means that the calculation failed, thus the quantifier becomes non-executable. In such a case, the function $\mathcal{C}_Q$ produces Java code that throws angelic undefinedness so that it be contextually interpreted by the parent expressions.

If the quantified variable's type is an integral type like `int`, then, instead of a set, an interval is calculated using the same technique. Interval patterns, e.g., x `<` E, are identified and appropriately accumulated based on logical connectives such as `||` and `&&`.

If the quantified variable's type is `boolean`, the quantifier is evaluated by explicitly enumerating all possible values, i.e., `true` and `false`.

If the quantified variable's type is a floating-point type such as `float` and `double`, the quantifier becomes non-executable; such a quantification cannot be evaluated by using the the static analysis approach described in this section.

Translation rules for other kinds of quantifiers and the set comprehension notation, such as the existential quantifier, general quantifiers, and the numerical quantifier, are defined in a similar fashion. In particular, they use the same technique and algorithm to compute the finite collections or intervals. Thus, their translation rules are not presented in this dissertation.

```
$\mathcal{C}[\![$(\forall $T$ $v$; $E_1$; $E_2$), $r$, $p]\!] \stackrel{\text{def}}{=}$ $\underline{\text{? } T \text{ is a reference type}}$
  try {
    $r$ = true;
    Collection $q$ = null;
    $\mathcal{C}_Q[\![E_1,\ v,\ q]\!]$
    Iterator $i$ = $q$.iterator();
    while ($r$ && $i$.hasNext()) {
      $T$ $v$ = ($T$)$i$.next();
      $\mathcal{C}[\![E_1$ ==> $E_2,\ r,\ p]\!]$
    }
  }
  catch (JMLAngelicException $e$) {
    $r$ = !$p$;
  }
  catch (Exception $e$) {
    $r$ = $p$;
  }
```

```
$\mathcal{C}_Q$: Expression $\times$ Identifier $\times$ Identifier $\rightarrow$ Program
$\mathcal{C}_Q[\![E,\ x,\ r]\!] \stackrel{\text{def}}{=}$
  ...
  $r$ = $\mathcal{Q}[\![E,\ x]\!]$ or throw new JMLNonExecutableException();
  ...
```

```
$\mathcal{Q}$: Expression $\times$ Identifier $\rightarrow$ $\mathcal{P}$(Program)
$\mathcal{Q}[\![E$.contains($x$), $x]\!]$  $\stackrel{\text{def}}{=}$  $\{E\}$  $\underline{\text{? if } E\text{'s type is } \texttt{Collection}}$
$\mathcal{Q}[\![E_1$ && $E_2,\ x]\!]$         $\stackrel{\text{def}}{=}$  $\mathcal{Q}[\![E_1,\ x]\!] \cap \mathcal{Q}[\![E_2,\ x]\!]$
$\mathcal{Q}[\![E_1$ || $E_2,\ x]\!]$         $\stackrel{\text{def}}{=}$  $\mathcal{Q}[\![E_1,\ x]\!] \cup \mathcal{Q}[\![E_2,\ x]\!]$
$\mathcal{Q}[\![E,\ x]\!]$                    $\stackrel{\text{def}}{=}$  $\top$   $\underline{\text{? all other patterns}}$
```

Figure 3.9: Translating the universal quantifier.

3.4 Related Work

3.4.1 Undefinedness

The undefinedness problem has been studied by many researchers in the framework of logics (e.g., [67] [121] [137]). The reason is that programming abounds with partial functions, and if partial functions are admitted in logical formulae, a programming logic is needed that handles partial functions and undefined expressions [137]. Two-valued logics with "undefined" handles partial functions by introducing a constant (e.g., $\perp$) to represent an undefined value [122]. Three-valued logics with "undefined" allows the logical formulae to have the undefined value [11] [14] [68] [79]. Some logics introduce even more than one special value, thus, leading to *many-valued logics* [9] [10]. Gries and Schneider modeled partial functions by under-specified total functions to avoid undefinedness and their approach keeps the logic simple and calculational [56]. JML adapts this semantics, and thus substitutes an arbitrary expressible value of correct type for undefined expressions [89, Section 3.1]. The motivating idea is to preserve the standard rules of logic in JML.

The simplest approach to undefinedness in runtime assertion checking would be to propagate to the user any exceptions thrown during the evaluation of assertions. The philosophy here is to view assertions as expressions of the underlying programming language, and no more or no less than that. An example is the assertion facility (`assert` statements) of the Java programming language [147] and most design-by-contract tools [5] [81] [109]. A shortcoming of this approach is that one cannot use the standard rules of logic to reason about assertions. As an example, a predicate `x.length > 0 || true` is not equivalent to `true`, where `x` is an array variable; i.e., `true` is not a zero element of disjunction[2]. The reason, of course, is that the variable `x` can be null. This approach would not be faithful to the semantics of JML, and it would not be definite about the top-level assertions.

A simple extension would be to "strictly" interpret undefinedness. An occurrence of an exception propagates, without being interpreted, up to the top-level assertions such as pre- and postconditions, where it is interpreted as `false`. That is, any exception in subexpressions falsify the top-level assertion. The runtime assertion checker now would have a definite answer for the top-level assertion, i.e., either `true` or `false`, and thus can take appropriate actions against assertion violations. However, a disadvantage of this approach is that, even if a subexpression does not contribute to the value of the whole assertion, it falsifies the whole assertion should it throw an exception. For example, if the variable `x` is null, a predicate `x.length > 0 || true` evaluates to `false`.

The problem of the above approach can be fixed by interpreting an exception non-strictly. The idea is to make the smallest boolean expression that encloses the exception-causing expression `false`, and thus not to propagate the exception any further. This non-strict interpretation lets more assertions be evaluated normally; i.e., all expressions enclosing the smallest boolean expressions now evaluates normally. The interpretation has a similar flavor as that of modeling partial functions by under-specified total functions [56]. This approach, however, has an unpleasant consequence. Consider a `requires` clause, `//@ requires !(x.length <= 0);`. If the variable `x` is null, the smallest, enclosing boolean expression `x.length <= 0` evaluates to false. This makes the whole assertion (i.e., the precondition) evaluate to true. Thus, the runtime assertion checker does not treat it as a precondition violation, missing an opportunity to detect a potential error.

The general problem in the non-strict approach is that an occurrence of an exception can contribute to make a top-level assertion true. This anomaly is caused by the fact the approach uses false as the default value for an exception. The problem was fixed by introducing the notion of contexts in the contextual interpretation presented in this chapter. The contextual approach is unique in that it implements Gries and Schneider's approach [56] in the framework of runtime assertion checking.

3.4.2 Quantified Expressions

There are two general schemes to evaluate quantified assertions: a *transformational approach* and a *generate-and-test approach*. In the transformational approach, a quantified assertion is translated into other languages, typically logic or constraint-based programming languages. The translated constraints are then solved using built-in constraint solving engines to evaluate the original quantification [153] [154]. The key concern here is to define translation rules that map from quantified assertions to the logical formula or constraints of the target programming languages so that the resulting formula or constraints be solved efficiently. The second is a direct and brute force approach. The underlying idea is to restrict quantifications to finite domains. If the domain that a quantified variable ranges over is known and finite, e.g., by restricting it to the elements of a collection or a finite interval of integral values, the quantified assertion can be evaluated for each element of the domain to determine the truth of the quantification [72] [81]. The first approach is popularly used in the context of specification execution, whose main goal is to execute, simulate, or animate specifications to find errors in specifications. However, it does not mesh well with the runtime assertion checking, where the target languages are imperative programming languages, such as C++, Eiffel, and Java. In JML, the approach would mean that assertion checking code rely on an external,

[2] Technically, the conditional-or operator (`||`) is not disjunction because it is short-circuit evaluated.

non-Java system, and violate one of the JML compiler's design goal — generating Java bytecode that works on any JVMs.

In the generate-and-test approach, there are at least three possibilities in achieving the finiteness of domains. The first is syntactic restriction. The syntax of quantified assertions may be restricted in such a way that the user must specify the domain of a quantified variable to be finite, typically a collection of objects or an interval of integral values [72] [81]. The second is static analysis. The syntax of quantified assertions is not restricted in anyway, but the assertions are analyzed statically to come up with finite domains that are sufficient to determine the truth of the quantifications [99] [152]. If no such finite domains are found, the assertions become non-executable. The third approach is based on the notion of *type extent*. The core idea here is to view a type as a collection of its instances [23]. In this view, a quantified variable ranges over all existing instances of its (declared) type, and the assertion is evaluated for each element of the type extent to determine the truth of the quantification [74]. While the last approach is based on a type point of view, the other two approaches are based on a collection point of view in that the domain is restricted to a (user specified) finite collection, and a quantified variable is bound to the element of the given collection rather than to the extent of its type [72].

Several researchers mentioned the idea of extending Eiffel with quantifications. McKim and Mondu's proposal appears to be one of the earliest [106]. Their main concern was to better document class interface design, and thus they only defined the syntax for universal and existential quantifications with no formal semantics or implementation hints. Walden and Nerson introduced a similar syntactic extension in the BON notation, an Eiffel-based object-oriented analysis and design method [155]. Kent and Maung took the type point of view in their proposal [74]. They distinguished two kinds of quantifications, value quantifications reference quantifications, and they also defined formal semantics for their quantifiers. For a value quantification, a quantification over the so-called *value types* [143] which intrinsically are immutable and lack identities, the type extent denotes all possible values. For a reference quantification, it denotes all currently existing objects. They also gave hints on implementing both kinds of quantifications. Katrib and Coira suggested a more practical and implementation-oriented approach [72]. According to our classification scheme, the approach took a collection point of view with syntactic restriction [73]. An iterator was introduced as a built-in language construct, and a quantified assertion was defined in terms of an iterator; the domain of a quantified variable must be an iterator.

As mentioned before, the type extent approach is unsound with respect to the semantics of JML, as it can produce false positives. Thus, the JML compiler adopts the static analysis approach, in particular the collection view point. It extends Lin's earlier work [99] by supporting more patterns and all kinds of quantifiers; Lin's work supports only simple quantifiers of the form, `(\forall` T v`;` c`.contains(`v`);` E`)`. The JML compiler can also handle pre-state expressions such as old variables and old expressions appearing inside quantifiers. In addition, the evaluation approach is integrated into the contextual interpretation framework for undefinedness.

There are several design-by-contract tools for Java that support quantified assertions [5] [81]. All of them syntactically restrict the domains of quantified variables to finite collections or intervals, e.g., by allowing only `java.lang.Enumeration`, `java.lang.Iterator`, or subranges of integral values. The general form of quantified assertions is `(forall` T x `in` D `|` $P(x)$`)`, where D is a finite domain and $P(x)$ is a predicate over the quantified variable x. In JML, no such restrictions are imposed on quantified expressions, thus JML's quantifiers may not be executable.

The Object Constraint Language (OCL) supports a limited form of quantifications [156] [157]. The OCL is a specification language for describing constraints on object-oriented models or systems with support for describing pre and postconditions and invariants. In OCL, quantifications are defined as operations on collection types with the general form, `aCollection-> forAll(`x`:`T`|`$P(x)$`)` and `aCollection->exists(`x`:`T`|`$P(x)$`)`, where $P(x)$ is a predicate expression involving the variable x. The quantified variable x is bound to each element of the collection `aCollection`, and the expression $P(x)$ is evaluated to determine the result of the quantified operation.

Some query languages for object-oriented database systems support very powerful quantification mechanism with both the type point of view and the collection point of view [23].

Peters and Parnas define quantified expressions in terms of so-called a *inductively defined predicate* [123]. Their idea is to restrict a quantification to a finite set by permitting only special forms of predicate expressions, i.e., $(x, P(x) \Rightarrow Q(x))$ and $(x, P(x) \wedge Q(x))$ for the universal and the existential quantifications, where $P(x)$ is an inductively defined predicate. The predicate $P(x)$ is a characteristic predicate of a finite set and must be given an inductive definition by specifying an initial finite set, I, a generation function, G, and a decision predicate, D. For example, the characteristic predicate of the set of integers from M to N, inclusive, is inductively defined by $I = \{M\}$, $G(x) = x + 1$, and $D(X) = x \leq N$. A set of C++ classes was written to execute inductively defined predicates. Inductively defined predicates are similar to iterators and loop constructs of programming languages, but formulated in predicates. Thus, their approach falls in the category of syntactically restricting quantifiers.

3.5 Discussion

3.5.1 Anomaly of Contextual Interpretation

The translation rule for equality expressions (`==`) has a couple of anomalies (see Figure 3.7 for the translation rules). One anomaly is that an occurrence of demonic undefinedness may not lead to an assertion violation, when expected. As a simple example, consider a precondition specified by `//@ requires (x.length > 0) == (y.length > 0);`. If both the variables `x` and `y` are null, then the evaluations of both the terms `x.length` and `y.length` complete abruptly by throwing `NullPointerException`. The 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 precondition evaluates to `true`, as it becomes equivalent to `false == false`, and thus no assertion violation is reported by the runtime assertion checker. However, since it is a source of a potential error, the runtime assertion checker would better report it as an assertion violation.

To handle this kind of situations, the JML compiler introduces a new context, called a *neutral context*, that may temporarily disable the contextual interpretation. If undefinedness occurs in a neutral context, it may be propagated to the parent expressions until it encounters a non-neutral context or a neutral context that is decisive. A neutral context is called *decisive* if it can interpret an occurrence of undefinedness as either true or false, under the standard rules of logic; otherwise, it is called *indecisive*. For example, a neutral context for an expression like `x.length > 0 || true` is decisive, as an occurrence of undefinedness in `x.length` does not contributes the value of the expression, and thus the expression becomes true even if it is evaluated in a neutral context. However, a neutral context for an expression like `x.length > 0` is indecisive, and it propagates any occurrences of undefinedness to its parent expression. Consider the previous precondition example again. Both the operand expressions, `x.length > 0` and `y.length > 0` are now evaluated in (indecisive) neutral contexts. Thus, any occurrence of `NullPointerException` reach the top-level, negative context; remember that a top-level assertion such as pre- and postconditions starts with a negative context. If both operands throw `NullPointerException`, the top level context can now detect and interpret it as false, and thus signalling a precondition violation.

The second anomaly is that the order of evaluating operand expressions may matter if one operand encounters an occurrence of angelic undefinedness and the other encounters an occurrence of demonic undefinedness. In the current rule, whichever becomes evaluated first gets the precedence. The JML compiler handle this by evaluating both operands and give a precedence to demonic undefinedness; the reason being that the runtime assertion checker can detect more errors in that way. The translation rule for a method call expression has the same problem, and the same technique is applied.

3.5.2 Referring to Pre-state Expressions

In JML, one can refer to pre-state expressions in post-state assertions such as postconditions and history constraints (see Section 4.6). 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 for evaluating the 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 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., `JMLAngelicException` or `RuntimeException`. Thus, a reference to a pre-state is contextually interpreted by the referring post-state assertion (see Section 7.8 and Section 8.4 for details)[3].

3.5.3 Other Approaches to Quantifiers

It is possible for the JML compiler to support the type extent-based approach. The JML compiler may compile classes in such a way that they maintain their own type extents. With self-maintained type extents, quantifiers can be evaluated by iterating over all elements in the type extents. However, the approach has a few shortcomings, some found through a prototype implementation in the JML compiler. As noted earlier, the approach is not sound with respect to the semantics of JML. It is not clear how to handle quantifiers with variables bound to interface types. Such variables may refer to an arbitrary instance of an arbitrary class that implements the interface. The approach is inefficient, as every object has to be registered into the type extent; worse, to support separate compilation, every class has to be compiled with type extent for its potential use in another type.

Another possibility would be to adapt some sort of statistical approaches, e.g., sampling for infinite domains. The sampling may be done randomly or using heuristics or boundary conditions found by static analysis on the quantified expressions. The statistical approach may even offer different levels of accuracy, e.g., a faster but less accurate check versus a slower but more reliable check. By their nature, the approaches are inaccurate, and unsound with respect to the semantics of JML. However, they could provide practical approximations for infinite domains, such as `float` and `double`, that the static analysis approach cannot currently handle.

3.6 Summary

The JML compiler handles the undefinedness problem by interpreting 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 is unique in that it implements, in runtime assertion checking, Gries and Schneider's approach [56] of viewing partial functions as under-specified total functions to avoid undefinedness. The approach is sound with

[3] An old variable needs a further treatment as the same variable may be used both in a negative context and a positive context. One solution would be for such a variable declaration to evaluate its pre-state 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 by passing polarity as an argument. This feature is not yet implemented by the JML compiler.

respect to the semantics of JML, and orthogonal to the various reasons of undefinedness, and thus can handle undefinedness caused by non-executable constructs. As the approach is not in any way JML-specific, it can be applied to other formal BISLs and design-by-contract tools.

The JML compiler provides an extensible framework to host multiple evaluation strategies for quantifiers. However, the primary strategy is a pattern-based static analysis approach that statically determines a set or interval of values that are enough to check at runtime to decide the truth of quantification; if no such set or interval is identified, the quantifier becomes non-executable, and thus interpreted contextually. The approach is incomplete in that not all quantified expressions are executable, but it is sound, practical, and efficient. However, it is found, through an empirical study, that quantifiers are most often written in such a way that their domains are restricted to finite collections or intervals.

Chapter 4

Method Specifications

This chapter explains how the JML compiler translates method specifications such as pre- and postconditions into runtime assertion checking code. A method specification is desugared into a simpler form suitable for automatic translation. A desugared top-level assertion such as pre- and postcondition is translated into a separate assertion checking method such as pre- and postcondition methods. A wrapper method is introduced to check method specifications by calling appropriate assertion checking methods; the wrapper method replaces the original method. In-line assertions such as `assert`, `assume`, and loop invariants and variants are translated into runtime assertion checking code blocks that are directly injected into the method body. Assertion violations are reported as special kinds of errors reserved by the runtime assertion checker. Assertion violation errors are organized into an exception hierarchy.

4.1 Introduction

In JML, the behavior of a method is specified in the Hoare-style pre and postconditions, but with many extensions and improvements (see Section 2.3). JML extensions and improvements include multiple clauses, defaults for omitted clauses, nested specifications, case analysis, old variables and expressions, redundant specifications, specification visibility, lightweight versus heavyweight specifications, concurrency specifications, termination specifications, example specifications, frame properties, subclassing contracts, and model programs. Of all these features, what should be checked at runtime? Most of these features are considered except for concurrency, termination, examples, frame properties, subclassing contracts, and model programs. Some features not considered such as frame properties and subclassing contracts can be checked at compile-time through static analysis [22] [135].

From a temporal point of view, JML constructs for method specifications are categorized into three groups:

- Pre-state specifications: These are specification that must be checked in the pre-state, i.e., just before the evaluation of the method body. Example constructs are `requires` clauses, `non_null` annotations to parameters, old variables, and old expressions. Note that old expressions appear in post-state assertions, but they must be evaluated in the pre-state.

- Post-state specifications: These are specification that must be checked in the post-state, i.e., just after the evaluation of the method body. Two such constructs are `ensures` clauses, `signals` clauses.

- Internal specifications: These are specifications that must be checked in the internal states, i.e., during the evaluation of the method body. Examples are in-line assertions such as `assert`, assume, hence_by, unreachable, set statements, and loop invariants and variants.

JML makes a distinction between lightweight specifications and heavyweight specifications (see Section 2.3.2). Does this distinction matter to the runtime assertion checker? A heavyweight specification is complete, thus its runtime checking should be complete, modulo non-executable specifications. In particular, an omitted clause either trivially holds or trivially fails, as it defaults to either true or false[1]. A lightweight specification may be incomplete, thus its runtime checking may be incomplete. In particular, an omitted clause does not contribute assertion checking, as it default to \not_specified. For example, if a lightweight specification has no requires clause, the runtime assertion checker performs no precondition checking. In sum, the runtime assertion checker should handle omitted clauses based on the specification style. The difference of specification styles shows up when method specifications are desugared before being translated into runtime assertion checking code (see Section 4.2).

JML has a notion of *specification privacy* that allows one to write several specifications of different privacy levels for methods (see Section 2.3.4). The idea is to limit the visibility of the specification to a particular kind of clients, thus to give different specifications to different clients. The question is then specifications of what privacy levels should be checked. Should all specifications of the method be checked, or specifications of particular privacy levels be checked, e.g., depending on the caller? As discussed in Section 1.4.1, the answer depends on the role of the runtime assertion checker. In this dissertation, the runtime assertion checker is viewed as a debugging and testing tool, and thus it checks specifications of all privacy levels, regardless of the caller. Interestingly, a modular verification requirement imposes certain restrictions in the way specifications of different privacy levels are written, and if the requirement is satisfied, the runtime assertion checker blames contract bleaches correctly; this also means that the runtime assertion checker can be used to check whether specifications satisfy the modular verification requirement (see Section 4.8).

4.1.1 Reporting Assertion Violations

If there is an assertion violation, the runtime assertion checker has to let the user know it. The question is how to report an assertion violation to the user. There are several possibilities, including a drastic approach like aborting the program, printing or logging messages, and throwing an exception indicating an assertion violation. The exception approach suits the runtime assertion checker best, as it provides a programmable solution in that that tools (including the checker itself) can catch an assertion violation exception and perform an appropriate action [29]. For the approach to work, however, the checker has to reserve certain exception classes for reporting assertion violations. Another important question is whether an assertion violation should be an error in the sense of having the type java.lang.Error, or an exception by having the type java.lang.RuntimeException[2]. An assertion violation is viewed as a situation that programs should not attempt to recover from unless they are tools relying on the runtime assertion checker. The Java convention is to use an error class to indicate a serious problem that a reasonable application should not try to catch. Thus, assertion violations are defined as error classes.

Figure 4.1 shows the assertion violation errors of JML organized into an exception hierarchy. The abstract class JMLAssertionError, which is a subclass of class java.lang.Error, is the ultimate superclass of all assertion violation errors. This class has several subclasses that correspond to different kinds of assertion violations, such as precondition violations, postcondition violations, invariant violations, and so on. A precondition violation, JMLPreconditionError, is further distinguished into an entry precondition violation, JMLEntryPreconditionError, and an internal precondition

[1] Exceptions are specification clauses that are not based on predicates, such as accessible clauses and assignable clauses. However, such clauses are not considered in this dissertation.

[2] An assertion violation object should be a direct or indirect instance of the class java.lang.Error or java.lang.RuntimeException because in Java an exception thrown by a method should be an instance of one of above two classes or their subclasses.

- *JMLAssertionError*
  - *JMLPreconditionError*
    - JMLEntryPreconditionError
    - JMLInternalPreconditionError
  - *JMLPostconditionError*
    - JMLNormalPostconditionError
    - JMLExceptionalPostconditionError
  - *JMLIntraconditionError*
    - JMLAssertError
    - JMLAssumeError
    - JMLHenceByError
    - JMLLoopInvariantError
    - JMLLoopVariantError
    - JMLUnreachableError
  - JMLInvariantError
  - JMLConstraintError

Figure 4.1: Class hierarchy of assertion violation errors.

violation, `JMLInternalPreconditionError`. The first refers to a violation of a called method's precondition by a client, and the second refers to a precondition violation occurring while the body of the called method is being executed. This distinction is necessary for a tool to correctly assign blame upon assertion violations [29]. That is, the client must be blamed for an entry precondition violation while the method's implementation code must be blamed for an internal precondition violation. The class `JMLIntraconditionError` is for reporting violations of in-line assertions such as `assert` and `assume` statements (see Section 4.7). The classes `JMLInvariantError` and `JMLConstraintError` are for reporting violations of type invariants and history constraints respectively (refer to Chapter 5).

4.1.2 Translation Scheme

Method specifications are translated into runtime assertion checking code in three conceptual steps. These steps are introduced for the ease of explanation; the JML compiler combines and refines them further.

1. Simplify specifications into translatable forms. This step identifies and extracts executable assertions from method specifications, and then simplifies the identified assertions into forms amenable to automatic translation. The main part of simplification is desugaring of specifications [130] (see Section 4.2).

2. Translate desugared assertions into runtime assertion checking code. Each top-level assertions such as pre- and postconditions are translated by applying the translation rules defined in Chapter 3.

3. Attach the assertion checking code to the original code. This step is for injecting the assertion checking code into the appropriate place of the original code. For example, the pre- and postcondition checking code must be executed before and after the execution of the method body.

How is the assertion checking code injected into a method so that, for example, the method's pre- and postconditions are checked before and after the execution of the method body? Two possibilities are an in-line approach and a wrapper approach.

- *In-line approach*: The assertion checking code is inserted directly into the body of the method being checked. For example, the precondition checking code becomes the first statement (or a block of code) of the method body [13].
- *Wrapper approach*: The runtime assertion checking code becomes a separate method — e.g., pre- and postcondition checking methods [28]. A wrapper method, replacing the method under checking, forwards all client calls to the original method. The method forward call is preceded by a call to the precondition checking method, and is followed by a call to the postcondition checking method.

The in-line approach has two shortcomings. First, it is not trivial to inject assertion checking code of the post-state assertions such as normal and exceptional postconditions, invariants, and history constraints. The assertion checking code may not be added at the end, because the method body may have `return` statements. Second, the approach does not facilitate a modular way of implementing specification inheritance. The assertion checking code cannot be inherited by subclasses, as it is embedded into the method body. For subclasses, assertion checking code must be regenerated or textually copied down from superclasses and implemented interfaces (which may need renaming and other modifications). On the other hand, the approach is simple and efficient; it does not incur extra method calls for assertion checking.

The wrapper approach is better structured and organized in that the instrumented code is modularized with wrapper methods and assertion checking methods. The approach also facilitates specification inheritance; a subclass can call the corresponding assertion checking methods of its superclasses to inherit specifications (refer to Chapter 6). Therefore, the JML compiler takes the wrapper approach, which is explained further in the following subsection.

4.1.3 Wrapper Approach

In the wrapper approach, specification assertions such as pre- and postconditions become separate *assertion checking methods*, also called *assertion methods* in short. A *wrapper method* replaces the method to be checked; thus, all client calls to the method now go to the wrapper method. The wrapper method is responsible for forwarding client calls to the original method with appropriate checks for assertions like pre- and postconditions. The assertions are checked by calling appropriate assertion checking methods (refer to Figure 4.2 and a detailed explanation below).

Two kinds of assertion methods are generated from method specifications: *precondition checking methods* and *postcondition checking methods*. A precondition checking method, also called a *precondition method* for short, checks the precondition of a method. If the precondition does not hold, the precondition method throws a precondition violation error. A postcondition checking method, also called a *postcondition method* for short, checks the postcondition of a method; if the postcondition does not hold, it throws a postcondition violation error. There are two kinds of postcondition methods: a *normal postcondition method* and an *exceptional postcondition method*. The normal postcondition method checks the normal postcondition of a method, specified by `ensures` clauses. The exceptional postcondition method checks the exceptional postcondition of a method, specified by `signals` clauses. The pre- and postcondition methods are explained in detail in Section 4.4 and Section 4.5 respectively.

Each method will have one precondition method, one normal postcondition method, and one exceptional postcondition method. Multiple specification clauses of the same kind are desugared and combined into one specification clause (see Section 4.2 for specification desugaring). A redundant specification such as `requires_redundantly` clauses may be also desugared and combined for translation[3].

Figure 4.2 shows the control flow in a wrapper method. A client call to the original method goes to the wrapper method. The wrapper method first calls the precondition method to check the

[3] A user can control this behavior of the JML compiler by supplying an appropriate command-line option.

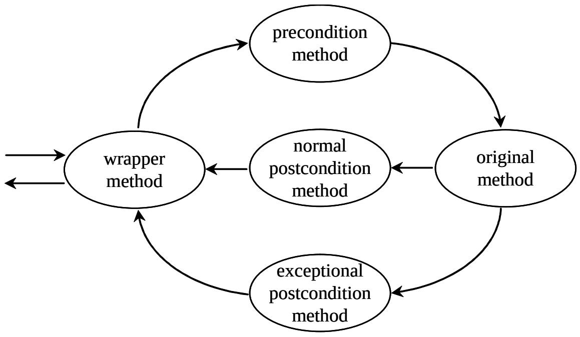

Figure 4.2: Control flow of the wrapper approach.

precondition, and then it calls the original method. If the original method terminates normally, the wrapper method calls the normal postcondition method; otherwise, i.e., if the method terminates abruptly by throwing an exception, the wrapper method calls the exceptional postcondition method. Each assertion method throws an assertion violation error if the corresponding assertion does not hold when it is called.

4.1.4 Outline

The rest of this chapter is organized as follows. Section 4.2 explains how method specifications are desugared into a simpler form to prepare for the automatic translation. Section 4.3 explains the structure of wrapper methods that replace the original methods to transparently check method specifications such as pre- and postconditions. Section 4.4 explains how preconditions are translated into precondition methods. Section 4.5 explains both normal and exceptional postcondition methods. Section 4.6 explains how the JML compiler handles pre-state assertions such as old variables and expressions, which are used in the post-state assertions but must be evaluated in the pre-state. Section 4.7 discusses in-line assertions such as `assert` and loop invariants and variants. In-line assertions are translated into assertion checking blocks. Section 4.8 discusses modular verification requirements and checking constructors and initializers.

4.2 Desugaring Specifications

JML provides a rich set of constructs for writing method specifications, such as multiple clauses, nested specifications, and case analysis (see Section 2.3). A method specification is desugared into a simpler form, before it is translated into runtime assertion checking code. The main purpose of desugaring here is to facilitate translating method specifications; thus, it differs in several ways from desugaring for semantics [107] [130]. For example, specification clauses that are not translated into runtime checks, such as `assignable` clauses and `measured_by` clauses, are ignored. The privacy levels of specifications are also ignored, as the runtime assertion checker checks specifications of all privacy levels (see Section 4.1). Consequently, the meaning of a desugared specification may not be equivalent to the original specification in the standard JML semantics, but it is equivalent from the operational view point of the runtime assertion checker. This section explains aspects that are unique in desugaring for translation into runtime assertion checking code.

A method specification, M, is viewed as a sequence of specification cases, $C_{i=1,\ldots,n}$. Each *specification case*, C_i, is a tuple, $\langle V_i, P_i, Q_i, R_i \rangle$, consisting of a sequence of old variable declarations, V_i, a set of `requires` clauses, P_i, a set of `ensures` clauses, Q_i, and a set of `signals` clauses, R_i. Using the desugaring of Raghavan and Leavens [130], specification cases with nested specifications or case analysis are assumed to be already desugared into a flat specification (refer to Section 2.3). A specification case can be either lightweight or heavyweight.

A method specification, $M = \langle V_i, P_i, Q_i, R_i \rangle_{i=1,\ldots,n}$, is desugared into a pair, $\langle D, M' \rangle$, consisting of a sequence of variable declarations, $D = \langle v_i, E_i \rangle_{i=1,\ldots,m}$, and a new, desugared method specification, $M' = \langle \epsilon, P'_i, Q'_i, R'_i \rangle_{i=1,\ldots,n}$. The variable declarations, D, are for handling references to pre-state expressions (e.g., old variables and old expressions) appearing in post-state assertions (see Section 2.3 for pre-state expressions). The idea is to evaluate each pre-state expression, E_i, in the pre-state, assigning the result to the corresponding fresh variable, v_i, and to substitute in the post-state assertions the variable v_i for each reference to the corresponding pre-state expression (see Section 4.6). Thus, each variable declaration, $\langle v_i, E_i \rangle$, introduces a fresh variable, v_i, with an associated pre-state expression, E_i. Each pre-state expression, E_i, is either the expression part of an old variable declaration or an old expression appearing in `ensures` or `signals` clauses of the original method specification, M. The variable declarations, D, are built as follows.

1. For each old variable declaration[4], $\langle o_j, E_j \rangle$, in V_i, a new variable declaration, $\langle v_j, E_j \rangle$, is added to D, where v_j is a fresh variable. Note that a unique variable is introduced for each old variable because different specification cases may declare an old variable of the same name.

2. For each old expression, O_j, appearing in Q_i and R_i, a new variable declaration, $\langle v_j, E_j \rangle$, is added to D, where v_j is a fresh variable.

The desugared method specification, $M' = \langle \epsilon, P'_i, Q'_i, R'_i \rangle_{i=1,\ldots,n}$ has the following properties.

1. Each specification case, $\langle \epsilon, P'_i, Q'_i, R'_i \rangle$, is a lightweight specification. A heavyweight specification is transformed into a lightweight specification.

2. No assertions of each specification case, $\langle \epsilon, P'_i, Q'_i, R'_i \rangle$, refer to the declared old variables. That is, P'_i, Q'_i, and R'_i have no occurrences of old variables.

3. No post-state assertions of each specification case, $\langle \epsilon, P'_i, Q'_i, R'_i \rangle$, have old expressions. That is, both Q'_i and R'_i have no old expressions.

4. Each occurrence of the reserved word `\result` in post-state assertions is replaced with the name `rac$result`. The name `rac$result` is a formal parameter used to pass the result of the original method to the postcondition methods (see Section 4.5).

The first property is for the convenience of translation. With this property, all specification cases can be treated uniformly regardless of their specification style. However, an implication is that a heavyweight specification must be completely desugared, as the runtime assertion checker treats it as a lightweight specification. In a lightweight specification, an omitted clause is interpreted as being not specified; e.g., an omitted `requires` clause defaults to `requires \not_specified;`. Thus, the runtime assertion checker does nothing for an omitted clause; it cannot do anything for clauses that are not specified. However, this behavior would be wrong for a heavyweight specification, because in a heavyweight specification an omitted clause defaults to true; e.g., an omitted `requires` clause defaults to `requires true;`. In addition, short-hand forms of heavyweight specifications such as normal and exceptional specifications (`normal_behavior` and `exceptional_behavior`) must be desugared into the general behavior form (`behavior`). This is done by using Raghavan and Leavens's desugaring [130]. A fully-desugared heavyweight specification case is transformed into a lightweight specification case by simply dropping the behavior keyword `behavior`.

The second and third properties are for evaluating pre-state expressions appearing in the post-state assertions. This is done by substituting fresh variables for all such old variables and expressions; the fresh variables become the new variable declarations, D (see Section 4.2.1 and 4.2.2 below).

[4] In JML, an old variable declaration has the form, `old` T o `=` E`;` (refer to Section 4.6 for an example).

A desugared method specification, $M' = \langle \epsilon, P'_i, Q'_i, R'_i \rangle_{i=1,\ldots,n}$, with a sequence of fresh variable declarations, $D = \langle v_i, E_i \rangle_{i=1,\ldots,m}$, is translated into assertion checking methods. The precondition P_i's are translated into a precondition method (see Section 4.4), the normal postcondition Q_i's into a normal postcondition method (see Section 4.5), and the exceptional postcondition R_i's into an exceptional postcondition method (see Section 4.5). The pre-state expressions E_i's of the variable declarations, D, are evaluated by the precondition method to set the corresponding fresh variables v_i's which become private fields (see Section 4.6).

4.2.1 Eliminating Old Variables

Desugaring eliminates all references to old variables from specification cases (see Section 4.6 for an example of old variables). Suppose a specification case, $C = \langle V, P, Q, R \rangle$. Let the old variable declarations, V, be a sequence of variable declarations, $\langle o_i, E_i \rangle_{i=1,\ldots,n}$, where o_i is an old variable and E_i is the JML expression that defines the value of o_i, i.e., `old` T o_i `=` E_i`;`. Then, the desugared specification case, $C' = \langle \epsilon, P', Q', R' \rangle$, with new variable declarations, $D = \langle v_i, E'_i \rangle_{i=1,\ldots,n}$, where v_i is a fresh variable, is defined as follows.

$$P' \overset{\text{def}}{=} P[o_i := v_i], \;\; i = 1, \ldots, n \tag{4.1}$$

$$Q' \overset{\text{def}}{=} Q[o_i := v_i], \;\; i = 1, \ldots, n \tag{4.2}$$

$$R' \overset{\text{def}}{=} R[o_i := v_i], \;\; i = 1, \ldots, n \tag{4.3}$$

$$E'_i \overset{\text{def}}{=} E_i[o_j := v_j], \;\; i = 1, \ldots, n, \; j = 1, \ldots, i-1 \tag{4.4}$$

The notation $P[o_i := v_i]$ means replacing every free occurrence of o_i in P with v_i. The definition (4.4) is for handling old variables appearing in the declarations of other old variables.

There are two reasons for all old variables being replaced with fresh variables. First, it allows the runtime assertion checker to evaluate the associated expressions in the pre-state and to use their values in post-state assertions, such as normal and exceptional postconditions. Second, by introducing fresh variables, potential name clashes are prevented; more than one specification case may declare old variables of the same name.

4.2.2 Eliminating Old Expressions

Desugaring also eliminates all old expressions from specification cases, i.e., from normal and exceptional postconditions. The approach is similar to eliminating old variables. Let $C = \langle \epsilon, P, Q, R \rangle$ be a specification case with old expressions E_i's in the postconditions Q and R, where $i = 1, \ldots, n$. Then, the desugared specification case, $C' = \langle \epsilon, P, Q', R' \rangle$, with new variable declarations, $D = \langle v_i, E_i \rangle_{i=1,\ldots,n}$, where v_i is a fresh variable, is defined as follows.

$$Q' \overset{\text{def}}{=} Q[E_i := v_i] \;\; i = 1, \ldots, n \tag{4.5}$$

$$R' \overset{\text{def}}{=} R[E_i := v_i] \;\; i = 1, \ldots, n \tag{4.6}$$

Each old expression E_i in the postconditions Q and R is replaced with the corresponding fresh variable v_i.

4.3 Wrapper Methods

A wrapper method has the same name and signature as the method that it wraps with assertion checking code. The original method becomes a private method with a new, uniquely generated name. Figure 4.3 shows the general structure of a wrapper method, say, for a method m declared in a type S. Remember that the wrapper method's responsibility is to check assertions transparently. The wrapper method first checks pre-state assertions such as preconditions and invariants, if any, by calling appropriate assertion methods, e.g., `checkInv$`S and `checkPre$`m`$`S (see Section 4.4 for precondition methods and Section 5.2 for invariant methods). As mentioned earlier, the called assertion methods throw assertion violations errors (e.g., `JMLInvariantError`) if the assertions do not hold. Therefore, pre-state assertion violations result in appropriate assertion errors being thrown.

If all pre-state assertions hold, the now-renamed, original method is invoked in a `try` statement and the result, if exists, is stored into a temporary local variable. The stored result is used later to check postconditions. If the call to the original method does not result in an exception being thrown, it means that the called method terminates normally, establishing a normal post-state. Thus, the wrapper method checks the normal postcondition by calling the normal postcondition method, `checkPost$`m`$`S (see Section 4.5 for postcondition methods).

The first `catch` clause is for converting entry precondition violations into internal precondition violations. If the call to the original method results in an entry precondition violation, it is converted into an internal precondition violation. From the client's view point, such a precondition violation is an internal precondition violation, as it happened while executing the body of the original method; the original method might have called another method (including itself) whose preconditions were not met.

The second `catch` clause prevents assertion violation errors from being caught by the third `catch` clause that checks the exceptional postcondition. It propagates all assertion violation errors to the client.

The third `catch` clause identifies an abrupt completetion of the original method by catching all exceptions[5]. If the control reaches here, it means that the execution of the original method has thrown an exception, thus resulting in an exceptional post-state. Therefore, the exceptional postcondition is checked here by calling the exceptional postcondition method, `checkXPost$`m`$`S. To make assertion checking transparent, the exceptional postcondition method re-throws the original exception if the exceptional postcondition is satisfied (see Section 4.5).

Finally, the `finally` block checks the rest of post-state assertions such as invariants and history constraints (see Section 5). These checks are done only if no precondition and postcondition violations occurs earlier; the detailed mechanism is suppressed. As shown, invariants are checked both in the pre-state and post-state, but history constraints are checked only in the post-state.

The third `catch` clause raises a question. What if the execution of the original method encounters errors other than assertion violation errors? The semantics of JML is to release the implementation from its obligation to fulfill postconditions when the Java Virtual Machine (JVM) encounters an error [128]. In particular, JML states the following [89, Section 2.1.3.3].

> In general, a method specified with `normal_behavior` has a correct implementation if, whenever it is called in a state that satisfies its precondition, either the method terminates normally in a state that satisfies its postcondition, having assigned to only the locations permitted by its assignable clause, or Java signals an error, by throwing an exception that inherits from `java.lang.Error`.

The runtime assertion checker deviates from the standard semantics of JML by checking exceptional postconditions explicitly written in terms of `java.lang.Error` and its (user-defined) subclasses, e.g., `signals (MyError e) P;`. The reason for this is that if one writes such an exceptional postcondition, one would expect the runtime assertion checker to check it. For this, the wrapper method catch all such errors and invokes the exceptional postcondition method (see Section 4.5.2 for exceptional postcondition methods).

[5] In Java, the class `Throwable` is the superclass of all errors and exceptions. The class `Throwable` has two subclasses, `Error` and `Exception`. The first indicates serious problems that a reasonable application should not try to catch, whereas the second indicates conditions that a reasonable application might want to catch.

```
T m(T1 x1, ..., Tn xn) {
  checkInv$S();
  checkPre$m$S(x1, ..., xn);
  T rac$result;
  try {
    rac$result = orig$m(x1, ..., xn);
    checkPost$m$S(x1, ..., xn, rac$result);
    return rac$result;
  }
  catch (JMLEntryPreconditionError rac$e) {
    throw new JMLInternalPreconditionError(rac$e);
  }
  catch (JMLAssertionError rac$e) {
    throw rac$e;
  }
  catch (Throwable rac$e) {
    checkXPost$m$S(x1, ..., xn, rac$e);
  }
  finally {
    if (/* no postcondition violation? */) {
       checkInv$S();
       checkHC$S();
    }
  }
}
```

Figure 4.3: General structure of wrapper methods.

An assertion can contain calls to methods that have their own specifications. The wrapper method recognizes this situation and does not check a method's specification if the method is being called while checking assertions of another method (including the method itself). These details are suppressed in the code shown.

4.4 Precondition Methods

The precondition method for a method, say $m$, checks $m$'s precondition. If the precondition does not hold, the precondition method indicates this by throwing a precondition violation error. If the precondition holds, the precondition method produces no client-visible side-effects, provided that the precondition itself has no side-effects[6]. However, the precondition method makes certain side-effects for other assertion methods, e.g., postcondition methods. In particular, it stores the results of evaluating each specification case's postcondition so that the postcondition methods can refer to them when evaluating the corresponding postcondition in the post-state (see also Section 4.4).

Let $m$ be declared in type S, with signature T $m$($T_1\, x_1, \ldots, T_n\, x_n$) `throws` E_1, ..., E_m. Let $m$ have a desugared method specification, $\langle \epsilon, P_i, Q_i, R_i \rangle_{i=1,\ldots,k}$, where each $\langle \epsilon, P_i, Q_i, R_i \rangle$ is a specification case with the precondition P_i, the normal postcondition Q_i, and the exception postcondition R_i (see Section 4.2 for desugaring). Then, the precondition method for $m$, `checkPre$`m`$`S, has the general structure shown in Figure 4.4[7]. The notation $[\![P_i,\ \text{rac\$pre}_i]\!]$ is used to denote

[6] The JML type checker ensures that JML assertions have no side-effects by performing *purity checking*.

[7] The JML compiler uses the character `$` in all generated names such as method names to avoid a potential name

```
public void checkPre$m$S(T_1 x_1, ..., T_n x_n) {
  boolean rac$v = false;
  [[P1, rac$pre_1]]
  rac$v = rac$v || rac$pre_1;
  ...
  [[Pk, rac$pre_k]]
  rac$v = rac$v || rac$pre_k;
  if (!rac$v) {
    throw new JMLEntryPreconditionError(/* ... */);
  }
}
```

Figure 4.4: General structure of precondition methods.

the result of translating predicate P_i into Java program code; it is a short-hand notation for $\mathcal{C}[\![P_i, \texttt{rac\$pre}_i, \texttt{false}]\!]$ (see Chapter 3 for the translation function $\mathcal{C}[\![\cdot]\!]$. The translated code evaluates the predicate P_i's and stores the results into `rac$prei`'s, new private boolean fields introduced by the JML compiler. The fields are used by postcondition methods to check normal and exception postconditions corresponding to the precondition P_i's, i.e., Q_i's and R_i's respectively (refer to Section 4.5 for details). This is because the postconditions Q_i's and R_i's need to hold in the post-state only if their corresponding preconditions P_i's hold in the pre-state.

In essence, the precondition method **checkPre$m$S** evaluates each precondition P_i and disjoin the results. If the disjoined result becomes false, it throws a precondition violation error. As a side effect, the method stores the value of each precondition P_i into a new private field **rac$prei**.

However, there are some technical details. The precondition method is declared public to support specification inheritance implemented by using Java's reflection facility (see Chapter 6 for specification inheritance). For the same reason, precondition methods have unique names with their declaring type names as postfix, e.g., `checkPre$m$S`; this prevents the precondition method of a subclass' overriding method from overriding the precondition method of its superclass' overridden method.

4.5 Postcondition Methods

The postcondition methods for a method, say $m$, $m$'s prostonditions. There are two kinds of postcondition methods: a *normal postcondition method* and an *exceptional postcondition method*. The first checks the normal postcondition specified by `ensures` clauses, and the second checks the exceptional postcondition specified by `signals` clauses.

As in the previous section, let $m$ be declared in type S, with signature $T\ m(T_1\, x_1, \ldots, T_n\, x_n)$ `throws` E_1, ..., E_m. Let $m$ have a desugared method specification, $\langle \epsilon, P_i, Q_i, R_i \rangle_{i=1,\ldots,k}$, where each $\langle \epsilon, P_i, Q_i, R_i \rangle$ is a specification case with the precondition P_i, the normal postcondition Q_i, and the exception postcondition R_i

4.5.1 Normal Postconditions

The normal postcondition method for $m$, `checkPost$`m`$`S, the method $m$, checks the method's normal postcondition; if the normal postcondition does not hold, the method throws a normal postcondition violation error. Figure 4.5 shows the general structure of normal postcondition methods.

clash. The Java Language Specification suggests to use the `$` character only in mechanically generated code [55, Section 3.8].

```
public void checkPost$m$S(T_1 x_1, ..., T_n x_n, T rac$result) {
  boolean rac$v = true;
  if (rac$v && rac$pre_1) {
     ⟦Q_i, rac$v⟧
  }
  ...
  if (rac$v && rac$pre_k) {
     ⟦Q_k, rac$v⟧
  }
  if (!rac$v) {
    throw new JMLNormalPostconditionError(/* ... */);
  }
}
```

Figure 4.5: General structure of normal postcondition methods.

The normal postcondition method takes an additional argument, **`rac$result`**, through which the wrapper method passes in the return value of the original method $m$ (see Section 4.3). As each occurrence of `\result` in the postconditions Q_i's is desugared into `rac$result` (see Section 4.2), each `\result` in the postconditions effectively refers to the return value of the original method $m$.

As before, the notation $[\![Q_i,\ \texttt{rac\$}v]\!]$ denotes a sequence of Java statements to evaluate the predicate Q_i and store the result to the variable $\texttt{rac\$}v$ (refer to Chapter 3 for the translation of expressions). In the body, the normal postcondition of the method $m$ is checked. For this, each normal postcondition, Q_i, is evaluated, and the results are conjoined. If the conjoined result becomes false, the method throws a normal postcondition violation error; otherwise, the normal postcondition method has no effects visible to the client. The normal postcondition, Q_i, is evaluated only if the corresponding precondition, P_i holds; the semantics of JML is that the postcondition Q_i has to be satisfied only if the precondition P_i holds, i.e., $P_i \Rightarrow Q_i$. Remember that the value of the precondition P_i is stored into the private field `rac$prei` by the precondition method (refer to Section 4.4). In sum, the normal postcondition method evaluates the predicate $\bigwedge(P_i \Rightarrow Q_i)$.

4.5.2 Exceptional Postconditions

The exceptional postcondition method for $m$, **`checkXPost$m$S`**, checks $m$'s exceptional postcondition. If the exceptional postcondition does not hold, the method throws an exceptional postcondition violation error. Let the exceptional postcondition, R_i, of each specification case, $\langle \epsilon, P_i, Q_i, R_i \rangle$, consist of several `signals` clauses of the form `signals` $(X_{ij}\ e_{ij})\ R_{ij}$`;`, where X_{ij} is an exception type. That is, the method $m$'s specification has the form shown in Figure 4.6.

The semantics of JML states that each R_{ij} should hold if the method $m$ terminates exceptionally by throwing an exception and the thrown exception is of type X_{ij}. However, the implementation is released from its obligation to fulfill the exceptional postconditions if the method $m$ is called in a state where the precondition P_i does not hold.

Figure 4.7 shows the structure of $m$'s exceptional postcondition method, `checkXPost$m$S`. The code is presented by using a *literate programming* style notation to show long code structurally [78] [131]. In the code, for example, the text enclosed by a pair of ⟪...⟫, e.g., ⟪*checkCondition*⟫, is not a part of the code. It is a place holder marker to be substituted by a chunk of text that follows later and is preceded by ⟪...⟫ ≡, e.g., ⟪*checkCondition*⟫ ≡. The exceptional postcondition method has an additional formal parameter, **`rac$thrown`**, of type `java.lang.Throwable`. This is for getting the actual exception thrown by the original method $m$; the wrapper methods supplies an appropriate argument (refer to Section 4.3).

```
/*@  requires $P_1$;
  @  ensures  $Q_1$;
  @  signals ($X_{11}$ $e_{11}$) $R_{11}$;
  @  ···
  @  signals ($X_{1l}$ $e_{1l}$) $R_{1l}$;
  @ also
  @   ···
  @ also
  @  requires $P_k$;
  @  ensures $Q_k$;
  @  signals ($X_{k1}$ $e_{k1}$) $R_{k1}$;
  @  ···
  @ signals ($X_{kl}$ $e_{kl}$) $R_{kl}$; @*/
$T$ $m$($x_1$, ..., $x_n$) throws $E_1$, ..., $E_m$ { /* ... */ }
```

Figure 4.6: Method specifications with `signals` clauses.

The body consists of two parts, checking exceptional predicates, represented by the code chunk $\langle\!\langle \mathit{checkCondition} \rangle\!\rangle$, and rethrowing the original exception, represented by the code check $\langle\!\langle \mathit{rethrowException} \rangle\!\rangle$. The first part evaluates each exceptional postcondition R_i when the corresponding precondition P_i holds, and conjoins the results. If the conjoined result becomes true, the execution proceeds to the second part; otherwise, an exceptional postcondition violation error is thrown.. Remember that the value of the precondition P_i evaluated by the precondition method in the pre-state is found in the private field `rac$pre`$_i$ (see Section 4.4). In the nested `if` statements, each exceptional predicate R_{ij} is checked only if the exception thrown, `rac$thrown`, is of the corresponding exception type, X_{ij}. As before, the notation $[\![R_{ij}, \texttt{rac\$}v]\!]$ denotes a sequence of Java statements to evaluate the predicate R_{ij} and store the result to the variable `rac$`v (refer to Chapter 3 for the translation of expressions). The exceptional predicate R_{ij} may have an occurrence of a free variable e_{ij} that refers to the actual exception thrown. For this, the method declares a local variable with the same name and initializes it to the argument `rac$thrown`.

The second part of the body is for making assertion checking transparent to the client (see Figure 4.8). it is reached only if all exceptional postconditions are met, and re-throws the original exception thrown by the method m. A case analysis is performed to re-throw the original exception. If the exception thrown, `rac$thrown`, is of type `java.lang.RuntimeException` or `java.lang.Error`, it is re-thrown, if necessary, with an appropriate type casting. Otherwise, a similar form of testing and re-throwing is performed for each exception type X_i declared in the throws list of the method m. (In Java, an exception can be thrown by a method if only if it is of type `java.lang.RuntimeException`, `java.lang.Error`, or one of the method's declared exception types; this is statically checked by Java compilers and also by the JML typechecker.)

In sum, the exceptional postcondition method checks the effective exceptional predicate, $\bigwedge_i (P_i \Rightarrow \bigwedge_j Q_{ij})$. If the predicate does not hold, it throws an exceptional postcondition exception error; otherwise, it re-throws the original exception to make assertion checking transparent.

As discussed in Section 4.3, an implementation is released from its obligation to fulfill the exceptional postcondition when JVM signals an error by throwing an exception that inherits from `java.lang.Error`. Does the exceptional postcondition method satisfy this semantic requirement? It does unless there is a `signals` clause in the method specification whose exception type is `java.lang.Error` or its subclass. If there is no such a `signals` clause in the specification and the exception thrown is of type `java.lang.Error`, the exceptional postcondition is trivially satisfied. The code that evaluates each exceptional predicate R_{ij}, is never executed; it is executed only if the exception thrown, `rac$thrown`, is of the declared exception type, X_{ij}. Therefore, no assertion vi-

```
public void checkXPost$$m$$S$($T_1$ $x_1$, ..., $Tn$ $xn$, Throwable rac$thrown)
   throws $E_1$, ..., $E_m$ {
   $\langle\langle$checkCondition$\rangle\rangle$
   $\langle\langle$rethrowException$\rangle\rangle$
}

$\langle\langle$checkCondition$\rangle\rangle \equiv$
  boolean rac$$v$ = true;
  if (rac$$v$ && rac$pre$_1$) {
     if (rac$$v$ && (rac$thrown instanceof $X_{11}$)) {
        $X_{11}$ $e_{11}$ = ($X_{11}$) rac$thrown;
        $[\![R_{11}$, rac$$v]\!]$
     }
     ...
     if (rac$$v$ && (rac$thrown instanceof $X_{1l}$)) {
        $X_{1l}$ $e_{1l}$ = ($X_{1l}$) rac$thrown;
        $[\![R_{1l}$, rac$$v]\!]$
     }
  }
  ...
  if (rac$$v$ && rac$pre$_n$) {
     if (rac$$v$ && (rac$thrown instanceof $X_{k1}$)) {
        $X_{k1}$ $e_{k1}$ = ($X_{k1}$) rac$thrown;
        $[\![R_{k1}$, rac$$v]\!]$
     }
     ...
     if (rac$$v$ && (rac$thrown instanceof $X_{kl}$)) {
        $X_{kl}$ $e_{kl}$ = ($X_{kl}$) rac$thrown;
        $[\![R_{kl}$, rac$$v]\!]$
     }
   }
```

Figure 4.7: General structure of exceptional postcondition methods (part 1).

olation is thrown; instead, the original exception thrown, `rac$thrown`, is re-thrown by the second `if` statement of the second part of the body, and this makes assertion checking transparent to the client. If there is a `signals` clauses in the method specification whose exception type is `java.lang.Error` or its subclass, then the corresponding exceptional predicate is evaluated and checked. This deviates form the JML semantics, but it seems appropriate for runtime assertion checking. If one explicitly specifies such an exceptional postcondition, then one would be definitely interested in knowing the fact that the implementation does not meet the condition. In addition, the runtime assertion checker can be used to debug specifications and implementations with user-defined errors.

4.6 Pre-state Expressions

In method specifications, one can refer to pre-state values — expressions that should be evaluated in the pre-state — in the post-state assertions, such as normal and exceptional postconditions. In JML, there are two such mechanisms: old expressions and old variables. This section explains how the JML compiler supports such mechanisms for the postcondition methods to correctly refer to

```
⟪rethrowException⟫ ≡
  if (rac$thrown instanceof java.lang.RuntimeException) {
     throw (RuntimeException) rac$thrown;
  }
  if (rac$thrown instanceof java.lang.Error) {
     throw rac$thrown;
  }
  if (rac$thrown instanceof E_1) {
     throw (E_1) rac$thrown;
  }
  ...
  if (rac$thrown instanceof E_m) {
     throw (E_m) rac$thrown;
  }
```

Figure 4.8: General structure of exceptional postcondition methods (part 2).

```
try {
  ⟦O_i, rac$old_i⟧
} catch (JMLAngelicException rac$e) {
  rac$old_i = JMLRacValue.ofAngelic();
} catch (java.lang.Exception rac$e) {
  rac$old_i = JMLRacValue.ofDemonic();
}
```

Figure 4.9: Evaluating old expressions.

pre-state values.

An *old expression*, written as `\old(`E`)`, refers to the pre-state value of an expression, E. This is useful when specifying the behavior of a method with side-effects [8] [110]. Let $O = \{O_1, \ldots, O_n\}$ be the set of old expressions appearing in a method specification, i.e., in normal and exceptional postconditions Q and R. For each old expression O_i, the JML compiler generates a new private field, `rac$old`$_i$, of type `JMLRacValue` (see Section 4.2). The class `JMLRacValue` is a wrapper class to encapsulate values including undefinedness (refer to Section 3.2 for undefinedness). Remember from desugaring that, in the postconditions Q and R, each old expression O_i is replaced with the corresponding new field `rac$old`$_i$; the normal and exceptional postconditions become $Q[O_i :=$ `rac$old`$_i]$ and $R[O_i :=$ `rac$old`$_i]$ respectively. Therefore, postcondition methods refer to the new fields when evaluating the postcondition predicates.

The precondition method is responsible for setting each new field `rac$old`$_i$ to the value of the corresponding old expression, O_i, evaluated in the pre-state. For this, the precondition method is extended to include the code shown in Figure 4.9 for each old expression O_i: As before, the notation $[\![O_i,$ `rac$old`$_i]\!]$ denotes a sequence of Java statements to evaluate the expression O_i and store the result into the variable `rac$old`$_i$. The `catch` clauses are for propagating undefinedness from old expressions to postconditions.

An *old variable* is a specification variable that can be introduced into a method specification to abbreviate a pre-state expression. In the operation term, it means that the expression is evaluated in the pre-state and the result is bound to the variable. An old variable can be used in method specification clauses, such as `requires`, `ensures`, and `signals`. In Figure 4.10, for example, the

[8] An old expression can also be used in a history constraint (refer to Chapter 8 for details).

```
/*@ old int size = contents.size();
  @ requires size * 2 <= MAX_SIZE;
  @ ensures contents.size() == size * 2;
  @ signals (IllegalStateException e) size * 2 > MAX_SIZE;
  @*/
private void doubleSize() { /* ... */ }
```

Figure 4.10: Example of old variables.

variable `size` is an old variable that denotes the value of expression `contents.size()` in the pre-state.

The JML compiler handles old variables in the same way as old expressions. That is, for each old variable, it introduces a new private field. Each occurrence of an old variable is replaced with the new field whose value is initialized by the precondition method with the value of the associated expression evaluated in the pre-state.

With old variables and expressions taken into account, the precondition method has two responsibilities: to evaluate old variables and expressions to initialize the corresponding fields, and to check the precondition. The evaluation of old variables and expressions precedes the precondition checking, as the precondition itself may refer to old variables.

4.7 In-line Assertions

An *in-line assertion*, also called an *intracondition*, is an assertion that can be specified in the method body. In JML, an in-line assertion is treated as a statement, and thus can appear where a Java statement is allowed. JML provides several kinds of in-line assertions, such as `assert` statements, `assume` statements, `hence_by` statements, `unreachable` statements, `set` statements, and loop invariant and variant statements. This section explains how these in-line assertions are translated into runtime assertion checking code that is directly injected into the method body where the assertions are specified.

4.7.1 Assertions, Assumptions, and Reasons

Assertions — as formal facts or statements about the state of a program — are very useful for both debugging and proving the correctness of programs [160]. One knows that something is wrong with one's programs (or assertions) if assertions do not hold at runtime. In order to prove that a segment of code does what is intended, one needs to make assertions that are true at certain points in the code. JML's assertion statements are similar to Java's assertion statements [147] and assertion macros of C and C++ [36] [104] [134] [158] in that assertions are statements containing boolean expressions that the programmer believes to be true at the time the statements are executed. However, JML assertions are not limited to Java's expressions; assertions can be written by using quantifiers, model fields and methods, and other specification constructs. JML also makes a distinction between assertions and assumptions.

An `assert` statement is a specification statement containing a boolean expression that must hold when the control reaches the statement. An `assume` statement is a specification statement that specifies an assumption that the programmer makes on the program state when the control reaches the statement. An assertion is a condition that the implementation has to establish at that point of execution, while an assumption is not an obligation to the implementation. This distinction is useful in refinement calculus when specifications are viewed as contracts and implementations are viewed as refinements of contracts [4].

```
//@ assert P;
do {
  boolean rac$v = true;
  ⟦P, rac$v⟧
  if (!rac$v) {
    throw new JMLAssertError();
  }
} while (false);
```

```
//@ assume P;
do {
  if (JMLChecker.checkAssume()) {
    boolean rac$v = true;
    ⟦P, rac$v⟧
    if (!rac$v) {
      throw new JMLAssumeError();
    }
  }
} while (false);
```

Figure 4.11: Translating `assert` and `assume` statements.

How does this distinction affect runtime assertion checking? Apparently, one would like to know assertion violations, as assertions are the implementor's responsibility. What about assumptions? One would also like to know assumption violations, as subsequent assertions (and other specification statements) might rely on the violated assumptions. However, as assumptions are not the implementor's obligation, the JML compiler lets the programmer to tune the behavior of checking assumptions; assumption violation checking can be turn on and off by using runtime options.

Figure 4.11 shows how `assert` and `assume` statements are translated into runtime assertion checking code. The translated code evaluates the assertion or assumption predicate P and throws an appropriate intracondition violation error if the predicate does not hold. For assumptions, however, the predicate P is checked only if the method `JMLChecker.checkAssume()` returns true; the method tells if the programmer is interested in knowing assumption violations or not.

In addition to assertion and assumption, JML also provides `hence_by` statements that can specify the reasons or hints why one believes an assertion or an assumption holds. The `hence_by` statement has the form, ‘`//@ hence_by` P`;`”, where P is a predicate. The translation of `hence_by` statements is similar to that of assertion statements. The predicate, P, is evaluated and an intracondition violation error is thrown if it does not hold.

4.7.2 Unreachable Statements

An `unreachable` statement asserts that the statement should never be reached. This means that if the execution control ever reaches the unreachable statement, it is an assertion violation. Thus, the unreachable statement “`//@ unreachable;`”, is translated into Java code that simply throws an intracondition violation error.

```
do {
  throw new JMLUnreachableError(/* ... */);
} while (false};
```

The `throw` statement is wrapped with a `do while` statement not to interfere with the control flow analysis of Java compilers; without it, the statements following the generated `throw` statement become unreachable, threby, causing a compilation error.

4.7.3 Set Statements

A `set` statement is a specification statement that assigns a value to a ghost field. value of a ghost field. A *ghost field* is a field that can appear only in specifications (see Section 7.4 for ghost fields). The ghost field can be used in assertions as the same way as a Java program field is used in expressions. Figure 4.12 shows an example use of ghost fields and set statements. A static ghost

```
class GhostFieldAndSetStatementExample {
  //@ static ghost int counter = 0;
  GhostFieldAndSetStatementExample() {
    //@ set counter = counter + 1;
  }
}
```

Figure 4.12: Ghost fields and set statements.

```
//@ set g = E;
try {
  T rac$v;
  ⟦E, rac$v]]⟧
  ghost$g$S(rac$v);
}
catch (JMLAngelicException rac$e) {}
catch (java.lang.Exception rac$e) {}
```

Figure 4.13: Translating `set` statements.

field, `counter`, is used to keep track of the number of instances created, and its value is incremented in the constructor by using the `set` statement.

Figure 4.13 shows a sample translation of `set` statements. The ghost field $g$ is assumed to be declared in type S and have type T. The translated code evaluates the expression E and assigns the result to the ghost field. For the assignment, it uses the ghost field's setter method, `ghost$`g`$`S. (If the ghost field is declared in another type, then the setter method is called dynamically by using Java's reflection facility [146] to support separate compilation; these details are suppressed.) For each ghost field, the JML compiler generates a pair of getter and setter methods (see Section 7.4).

What happens if the evaluation of the expression, E, encounters undefinedness, e.g., exceptions or non-executable constructs? The current translation treats such cases as no operation provided that the expression E has no side-effects. Other alternatives are to assign the initial value, to report it as an error, or to handle contextually by assigning either demonic or angelic undefinedness. (see Section 3.2.3

4.7.4 Loop Invariants and Variants

An *invariant* is a condition that remains true during the execution of a segment of code. A *loop invariant*, a particular kind of invariants, is an assertion about a loop and remains true each time a loop condition is tested. This means that a valid loop invariant will be true the first time the loop begins execution, it will be true each time the loop starts execution just after the loop condition has been checked, and it will be true just after the loop is exited (after the condition has been checked). Thus, a loop invariant is a technique for proving some properties about a loop by expressing a condition that must be satisfied at the beginning of the loop, after every iteration and when the loop has terminated. Therefore, from the perspective of runtime assertion checking, a loop invariant must be checked (1) just before the loop body executes for the first time, (2) just after the loop body has executed for the last time, and (3) after every loop iteration in between.

In general, proving the termination of a loop is difficult, and a loop variant is one technique for proving the termination of a loop. A *loop variant* is an integer expression whose value is decreased upon each iteration of a loop but never below zero. Thus, a valid loop variant guarantees the termination of a loop. Therefore, the runtime assertion checker has to evaluate a loop variant

```
// original code
//@ maintaining I;
//@ decreasing V;
[L:] while (B)
      S

// translated code
{
  ⟪varDecl⟫
  [L:] while (true) {
    ⟪checkInv⟫
    ⟪checkVar⟫
    if (!(B)) {
      break;
    }
    S
  }
  ⟪checkInv⟫
}
```

```
⟪checkInv⟫ ≡
  boolean rac$v;
  ⟦I, rac$v⟧
  if (!rac$v) {
    throw new JMLLoopInvariantError();
  }

⟪varDecl⟫ ≡
  boolean rac$f = true;
  int rac$o = 0;

⟪checkVar⟫ ≡
  int rac$n = 0;
  ⟦V, rac$n⟧
  if (rac$n < 0 ||
      (!rac$f && rac$n >= rac$o)) {
    throw new JMLLoopVariantError();
  }
  rac$o = rac$o;
  rac$f = false;
```

Figure 4.14: Translating loop invariants and variants for `while` statements.

expression upon each iteration of the loop and make sure that the current value is greter than or equal to zero and less than the value of the previous iteration.

In Java, there are three kinds of iteration statements: `while` statements, `do` statements, and `for` statements. The remainder of this section explains how the JML compiler translates loop invariants and variants, annotated to each kind of Java iteration statements, into runtime assertion checking code.

In JML, a loop invariant statement starts with the keyword `maintaining` and a loop variant statement starts with the keyword `decreasing`. Both statements, if exist, have to proceed the loop statement that they annotate. For example, Figure 4.14 shows a skeleton `while` statement with the condition, B, and the body, S. The `while` statement is annotated with a loop invariant, I, which is a predicate, and a loop variant, V, which is an expression of type `int`. The notation [L:] denotes an optional label statement of the `while` statement.

The figure also shows the `while` statement translated by the JML compiler. The code chunk ⟪*varDecl*⟫ declares a couple of local variables to be used by the variant checking code, ⟪*checkVar*⟫, which is the translation of the variant, V. The code chunk ⟪*checkInv*⟫ checks the invariant, I. From the original `while` statement perspective, both the invariant and the variant are checked prior to the first and at the start of each iteration of the loop. After that, the loop condition, B, is tested. If the condition holds, the original loop body, S, is executed and then the next iteration begins; otherwise, the loop terminates by the `break` statement. When the loop terminates, the invariant is checked again. This last check ensures that the loop invariant holds even if the loop terminates abruptly or the condition, B, has side-effects (see below for a discussion on this).

The code that checks the loop invariant, I, is shown by the chunk ⟪*checkInv*⟫ ≡. It evaluates the invariant predicate, I, and throws a loop invariant violation error if the predicate does not hold.

The code for checking the loop variant, V, is shown by the chunk ⟪*checkVar*⟫ ≡, preceded by by a couple of local variable declarations. A loop variant should decrease upon each iteration of the loop. To check this, a local variable, `rac$`o, is introduced to stores the value of the loop variant in the previous iteration. A boolean variable, `rac$`f, is also used to to skip the loop variant check upon the

```
// original code
//@ maintaining I;
//@ decreasing V;
[L:] do {
  S
} while (B);
```

```
// translated code
{
  ⟪varDecl⟫
  [L:] while (true) {
    ⟪checkInv⟫
    ⟪checkVar⟫
    S
    if (!(B)) { break; }
  }
  ⟪checkI⟫
}
```

Figure 4.15: Translating loop invariants and variants for `do` statements. Refer to Figure 4.16 for the definitions of code chunk $\langle\!\langle \mathit{checkInv} \rangle\!\rangle$ and $\langle\!\langle \mathit{checkVar} \rangle\!\rangle$.

first iteration of the loop, as there is no earlier value. The loop variant check code, $\langle\!\langle \mathit{checkVar} \rangle\!\rangle \equiv$, evaluates the loop variant, V, and tests if the current value is greater than or equal to zero and also is less than the previous value; the second comparison is done only if it is not the first iteration. If the condition is not met, a loop variant violation error is thrown. Otherwise, local variables are updated to prepare for the next iteration.

It is possible that the evaluation of the loop variant, V, may encounter an exception or a non-executable construct. To cope with this, the whole expression that tests for the loop variant violation is evaluated contextually. That is, an exception is interpreted demonically while a non-executable construct is interpreted angelically (see Chapter 3). In addition, the variable `rac$`o is declared to be a special wrapper class, `JMLRacValue`, that wraps a Java value and undefinedness (see Section 4.6). These details are suppressed in the code shown.

The presented translation has several interesting properties. It can handle several forms of *abrupt completions* occurring during the execution of the loop body, S. If the statement, S, completes abruptly because of a `continue` statement with no label, then a new iteration begins in the instrumented code. As the loop invariant and the loop variant are checked at the beginning of each iteration, both will be checked as expected. However, an abrupt completion of S due to a labeled `continue` statement, in general, is not handled correctly. If the continue target is not the `while` statement, the loop terminates abruptly and thus the loop invariant is not checked. Similarly, the translation can correctly handle an abrupt completion caused by a `break` statement without a label, as the invariant is checked again upon the completion of `while` statement. For the same reason, a loop condition with side-effects is correctly handled by the translation. However, an abrupt completion of S due to a labeled `break` statement is handled incorrectly if the break target is not the `while` statement. The consequence is that one has to satisfy loop invariants and variants even if (1) the loop condition may have side-effects or (2) the loop statement terminates abruptly due to `continue` or `break` statements.

A `do` statement annotated with loop invariants and variants are translated in a similar fashion. Figure 4.15 shows a sample `do` statement and its translated code. The translated code has the exactly the same structure as that of the `while` statement except that the loop body, S, precedes the `if` statement that tests the loop condition, B.

The translation of `for` statements is slightly complicated because of loop initializer and update expressions, but the translated code has a similar structure to those of `while` and `do` statements As shown in Figure 4.16, suppose a `for` statement with initializer expressions, $I_1, ..., I_n$, the loop condition, B, update expressions, $U_1, ..., U_m$, and the body, S. The statement is annotated with a loop invariant, I, and a loop variant, V. In the translated code, the initializers, $I_1, ..., I_n$, are first executed outside the `while` statement. Thus, if the initializers declare any local variables, they will

```
// original code
//@ maintaining $I$;
//@ decreasing $V$;
[$L$:] for ($I_1$, ..., $I_n$;
        $B$;
        $U_1$, ..., $U_m$)
     $S$
```

```
// translated code
{
  boolean rac$$f$ = true;
  $I_1$, ..., $I_n$;
  [$L$:] while (true) {
    if (rac$$f$) {
        rac$$f$ = false;
    } else {
        $U_1$, ..., $U_m$;
    }
    ⟪checkInv⟫
    ⟪checkVar⟫
    if (!($B$)) { break; }
    $S$
  }
  ⟪checkInv⟫
}
```

Figure 4.16: Translating loop invariants and variants for `for` statements. Refer to Figure 4.16 for the definitions of code chunk ⟪*checkInv*⟫ and ⟪*checkVar*⟫.

be visible to the original loop body, S. For each iteration of the `while` loop except for the first, the update expressions, $U_1, ..., U_m$ is executed prior to the loop invariant, I, and variant, V, are checked and the body, S, is executed. The reason for making the update expressions to be the first, not the last statement of the `while` loop body is that the original loop body, S, may contains `continue` statements that transfer control to the beginning of the loop. In such an abrupt completion of S, however, the update expressions still have to be executed to prepare for the next iteration.

In this section, the translation of loop invariants and variants are explained by using a single loop invariant statement and a single loop variant statement. In JML, however, it is possible for a loop statement have more than one loop invariant and variant. The approach for multiple loop invariants is to combining them into a single invariant statement by conjoining the invariant predicates. For multiple loop variant statements, the approach is to have separate ⟪*checkInv*⟫ and ⟪*checkVar*⟫ for each loop variant statement.

4.8 Discussion

4.8.1 Privacy of Method Specifications

As discussed in Section 4.1, JML has a notion of specification privacy, and this raises an interesting question. Should specifications of different privacy levels of the same method be related in some way? For example, should the public specification imply the protected specification, the other way around, or in both direction? If one views method specifications as contracts, then for the precondition P, one has the following constraints, where P_{pub}, P_{pro}, P_{pkg}, and P_{pri} are public, protected, package-visible, and private preconditions.

$$P_{\text{pub}} \Rightarrow P_{\text{pro}} \tag{4.7}$$

$$P_{\text{pro}} \Rightarrow P_{\text{pkg}} \tag{4.8}$$

$$P_{\text{pkg}} \Rightarrow P_{\text{pri}} \tag{4.9}$$

Preconditions of wider visibility should be stronger than preconditions of narrower visibility. The reason for this is that a client would be surprised if encountered with assertion violation errors due to assertions not visible to the client, e.g., if P_{pro} is violated when the client calls the method in a state where P_{pub} holds.

For postconditions, one has the following constraint, where Q_{pub}, Q_{pro}, Q_{pkg}, and Q_{pri} are public, protected, package-visible, and private postconditions.

$$Q_{\text{pub}} \wedge Q_{\text{pro}} \Rightarrow Q_{\text{pkg}} \wedge Q_{\text{pri}} \tag{4.10}$$

The constraint is to support modular reasoning in the presence of specification inheritance. Consider a public method, m, declared in a class, T. When the method m is used in the class T, it would be okay to reason about the method call using all specifications of m, in particular, Q_{pri}, because all specifications of m are visible to such a method call. Now, consider a subclass, say S, of the class T. In the subclass S, however, only the public and protected specifications of m are visible. Therefore, if the class S wants to override the method m inherited from the class T, it can satisfy only m's public and protected specifications. Other specifications of m are not visible to S. Thus, the overriding method of the subclass S can establish only the postcondition $Q_{\text{pub}} \wedge Q_{\text{pro}}$. However, for the reasoning in the superclass T to be still valid when the overriding method of the class S is added to the system, one must have $Q_{\text{pub}} \wedge Q_{\text{pro}} \Rightarrow Q_{\text{pub}} \wedge Q_{\text{pro}} \wedge Q_{\text{pkg}} \wedge Q_{\text{pri}}$, which is simplified as $Q_{\text{pub}} \wedge Q_{\text{pro}} \Rightarrow Q_{\text{pkg}} \wedge Q_{\text{pri}}$. The implication of these constraints on the use of specification privacy needs to be further studied in the future.

4.8.2 Constructors and Finalizers

In Java, an explicit constructor invocation statement, if exists, should be the first statement of a constructor body [55, Section 8.8.5.1]. An explicit constructor invocation statement is a statement that calls another constructor of the same class (`this` call) or of the direct superclass (`super` call). This restriction poses a serious problem to correctly checking pre-states assertions (e.g., preconditions and lass invariants) of constructors. If a constructor has an explicit constructor invocation statement, then the statement must precede the calls to pre-state assertion methods in the constructor's wrapper constructor. Therefore, the pre-state assertions are not evaluated in the exact pre-state. A possible approach would be to generate Java bytecode directly without using intermediate source code. As Java bytecode does not enforce the above restriction, calls to pre-state assertion methods can be inserted into the correction position of a constructor body [82]. However, the consequence of such an approach needs to be studied further.

In Java, if a class contains no constructor declarations, a default constructor that takes no parameters is generated by the compiler [55, Section 8.8.7]. The default constructor invokes the superclass constructor with no arguments. The default constructor should also obey type assertions such as invariants and constraints (refer to Chapter 5 for type assertions). For this, the JML compiler generates a default constructor with appropriate assertion checking code if a class contains no constructor declarations.

A finazlier is treated like normal methods regarding runtime assertion checking of method specifications. That is, its pre- and postconditions are checked in the same way as in normal methods.

4.8.3 Initializers

An *initializer* is a block of code that is executed when an instance of a class is created (for an instance initializer) or the class is initialized (for a static initializer) [55, Section 8.6 and 8.7]. In JML, one can write method specifications for an initializer. The JML compiler does not support initializer specifications yet. However, the approach of checking method specifications may be adapted to check initializer specifications.

Chapter 5

Type Specifications

In this chapter, I explain how the JML compiler translates type specifications such as invariants and constraints into runtime assertion checking code. An invariant is a property on a single state, whereas a constraint is a property that may relate a state to an earlier state. Both assertions are desugared and then translated into assertion checking methods, called invariant methods and constraint methods respectively. The wrapper methods presented in the previous chapter are extended to call these assertion methods; thus, each method has to preserve invariants and constraints. I also explain how the specifications written in Java interfaces are translated.

5.1 Introduction

I use the term *type* to refer to both Java classes and interfaces. In JML, type specifications can have such JML annotations and declarations as invariants, history constraints, `represents` clauses, `depends` clauses, and specification-only declarations like ghost fields, model fields, model methods, and model types. An *invariant* is a property that should hold in all reachable states of an object, and a *history constraint* is a relation that should hold between each reachable state and all states that occur later in the program's execution [101] [103]. The term *type assertions* is used to refer to both invariants and constraints. The translation of type assertions into runtime assertion checking code is explained in this chapter, and specification-only declarations and related clauses such as `represents` clauses are discussed Chapter 7. This section discusses problems and a general approach to checking type assertions.

The following are problems and issues that need be addressed for checking type assertions.

- Instance versus static assertions: A type assertions may be either instance or static. An instance type assertion (e.g., an instance invariant) constrains only instance methods whereas a static type assertion (e.g., a static constraint) constrains both instance and static methods.

- Privacy of specifications: As in method specifications, type assertions have different visibility levels such as public, protected, package-visible, and private. Clients can see only assertions of appropriate visibility levels.

- Helper methods: A *helper method* (or constructor) is a private method that is exempted from the obligation of preserving type assertions. Helper methods are used to establish intermediate states that are not visible to clients.

- Old expressions and recursion: A history constraint may contain old expressions to refer to values in the earlier state. The old expressions in history constraints complicate runtime assertion checking if methods calls other methods (including themselves) of the same type. In

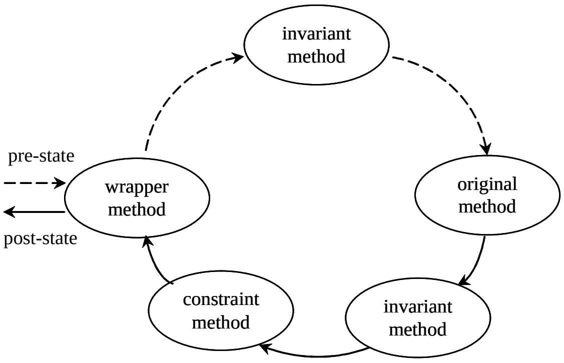

Figure 5.1: Control flow for checking type assertions.

every such call, the history constraints must be checked, the appropriate pre-states should be used to evaluate old expressions.

- Universal and method-specific constraints: In JML, a history constraint can be applicable to all methods or a set of named methods. I call the first a *universal constraint* and the second a *method-specific constraint.*

- Specifications for Java interfaces: Assertion methods cannot be added directly to Java interfaces; in Java, all methods in an interface must be abstract without method body.

As in the translation of method specifications, type assertions are translated into runtime assertion checking code in three conceptual steps. First, multiple type assertions of the same kind are conjoined and simplified to a single type predicate. Second, the conjoined predicate is translated into runtime assertion checking code by applying the translation rules for JML expressions (see Chapter 3). The resulting code becomes separate assertion methods such as invariant methods and constraint methods which are to be called by the wrapper methods (see Section 4.3 and 5.4 for wrapper methods). Third, the new assertion methods are added to the type.

Figure 5.1 shows the general control flow to check type assertions. Type assertions are checked whenever methods are called by clients. For this, the wrapper methods call both invariant and constraint methods. Invariants are checked both in the pre- and post-states while constraints are checked only in the post-state.

As mentioned earlier, JML distinguishes static and instance type assertions. A *static type assertion* specifies properties concerning the static fields of a type, whereas an *instance type assertion* specifies the instance fields; the former can be thought of constraining the type itself, while the latter can be though of constraining all objects of the type. To support this distinction, the JML compiler generates two kinds of assertion methods for type assertions: instance assertion methods and static assertion methods. The *instance assertion method* checks instance type assertions, and the *static assertion method* checks static type assertions. The wrapper methods call appropriate assertion methods depending on the `static`-ness of the methods. A static wrapper method calls only static assertion methods while an instance wrapper method calls both static and instance assertion methods.

As in method specifications, the runtime assertion checker checks type assertions of all privacy levels, the reason being that runtime assertion checking is viewed as a debugging tool. If type assertions of any privacy levels are violated by some method, then something is wrong with the method's implementation (or the type assertions themselves); one would be interested in knowing such cases to debug the implementation (or the specifications). This decision is also faithful to the

```
public void checkInv$instance$T() {
  boolean rac$v = false;
  ⟦P, rac$v⟧
  if (!rac$v) { throw new JMLInvariantError(); }
}

public static void checkInv$static$T() {
  boolean rac$v = false;
  ⟦Q, rac$v⟧
  if (!rac$v) { throw new JMLInvariantError(); }
}
```

Figure 5.2: Instance and static invariant methods.

semantics of JML, as all methods, regardless of their privacy levels, have to satisfy type assertions of all privacy levels.

Special treatment is needed for constructors and finalizers, as there may be no well-defined states to check type assertions for them. A constructor has to preserve all static type assertions, but in some cases instance type assertions become irrelevant to a constructor. For example, it is impossible to check instance constraints for constructors because there is no well-defined pre-state that can be related to the post-state. For the same reason, instance invariants are uncheckable in the pre-state of constructors. However, constructors have to establish instance invariants in the post-state. One exception is when they complete abruptly by throwing exceptions [82]. The reason for this exception is that exceptions are often used by constructors to indicate that they cannot perform their obligations, i.e., initializing new instances that satisfy the invariants. As a finalizer is a dual of a constructor, it needs not preserve instance type assertions in the post-state; however, it still has to preserve static type assertions in pre- and post-states.

As helper methods and constructors are exempted from the obligation of preserving type assertions, their wrapper methods do not call type assertion methods.

The rest of this chapter is organized as follows. Section 5.2 and Section 5.3 explain how invariants and constraints are translated into runtime assertion checking code. Section 5.3, in particular, examines problems and provides solutions for old expressions, recursions, and universal and method-specific constraints. Section 5.5 discusses an approach specific to specifications for Java interfaces, i.e., organizing assertion methods as separate assertion classes. Section 5.4 extends wrapper methods with respect to type assertions. Section 5.6 discuss limitations, establishment of static invariants, and implications of specification privacy to modular verification.

5.2 Invariants

In JML, one can separate invariants that are pertinent only to static fields and methods. Such invariants are called *static invariants.* A static invariant can refer to only static fields and methods; it cannot refer to instance fields or methods. However, an *instance invariant* can refer to both static and instance fields and methods. Thus, it constrains both static and instance states of program execution. Another difference is that instance invariants are inherited by subtypes, but static invariants are not (refer to Chapter 6 for inheritance of specifications).

For each kind of invariants, the JML compiler generates a separate invariant method, called a *static invariant method* and an *instance invariant method* respectively. Let T be a type with a set of instance invariants, $P_1, \ldots, P_n$, and a set of static invariants, $Q_1, \ldots, Q_m$. The invariants are first conjoined to form a single invariant predicate, i.e., $P \equiv P_1 \wedge \ldots \wedge P_n$, and $Q \equiv Q_1 \wedge \ldots \wedge Q_m$. The conjoined invariant predicates are then translated into instance and static invariant methods, whose

```
public void checkHC$instance$T() {
  restoreFrom$rac$stack();
  boolean rac$v = false;
  ⟦P, rac$v⟧
  if (!rac$v) { throw new JMLConstraintError(); }
}

public static void checkHC$static$T() {
  restoreFrom$rac$stack();
  boolean rac$v = false;
  ⟦Q, rac$v⟧
  if (!rac$v) { throw new JMLConstraintError(); }
}
```

Figure 5.3: Instance and static constraint methods.

general structures are shown in Figure 5.2 As before, the notation $[\![P,$ `rac$`$v]\!]$ denotes translated code that evaluates the predicate P and stores the result into the variable `rac$`v (refer to Chapter 3 for the translation of expressions). The invariant methods evaluate the conjoined invariant predicates and throw invariant violation errors if the predicates do not hold.

Invariant methods are called by wrapper methods (see Section 5.4).

5.3 Constraints

A (history) constraint is used to specify the way that objects can change their values over time [101] [103]. A method *preserves* a history constraint if the pre- and post-states are in the relation specified by the constraint. In JML, therefore, a constraint is typically written with old expressions (`\old(`e`)`) to relate the current state to the earlier state. JML distinguish *static constraints* and *instance constraints* [89]. An instance method has to preserve both static and instance invariants, whereas a static method has to preserve only static invariants.

How does the runtime assertion checker checks whether an object preserves its constraints? It views a constraint as a relation between the pre- and post-state of methods. The idea is to map the constraint's old expressions to the pre-state and other expressions to the post-state. Thus, constraints can be thought of as being implicitly conjoined to the postcondition.

As in invariants, the JML compiler generates separate constraint methods for static and instance constraints. Consider a type, T, with a set of instance constraints, $P_1, \ldots, P_n$, and a set of static constraints, $Q_1, \ldots, Q_m$. Multiple constraints are conjoined into a single constraint predicate, producing $P \equiv P_1 \wedge \ldots \wedge P_n$, and $Q \equiv Q_1 \wedge \ldots \wedge Q_m$. The conjoined predicates are then translated into constraint methods (see Figure 5.3). The constraint methods evaluate the conjoined constraint predicates and throw constraint violation errors if the predicates does not hold. The need for the first statement, `restoreFrom$rac$stack()`, is explained in Section 5.3.2.

Constraint methods are called by wrapper methods (see Section 5.4). An instance wrapper methods calls both static and instance constraint methods, and a static wrapper method calls only the static constraint method.

5.3.1 Old Expressions

A constraint predicate may contain old expressions that must be evaluated in the pre-state of method calls. How does the runtime assertion checker handle such old expressions? The approach is similar to handling old variables and expressions appearing in method specifications. The old expressions

```
public void evalOldExprInHC$instance$T() {
  try {
    T_1 rac$v;
    ⟦E_1, rac$v⟧
    rac$old_1 = JMLRacValue.ofObject(rac$v);
  }
  catch (JMLAngelicException rac$e) {
    rac$old_1 = JMLRacValue.ofAngelic();
  }
  catch (java.lang.Exception rac$e) {
    rac$old_1 = JMLRacValue.ofDemonic();
  }
  ...
  try {
    T_k rac$v;
    ⟦E_k, rac$v⟧
    rac$old_k = JMLRacValue.ofObject(rac$v);
  }
  catch (JMLAngelicException rac$e) {
    rac$old_k = JMLRacValue.ofAngelic();
  }
  catch (java.lang.Exception rac$e) {
    rac$old_k = JMLRacValue.ofDemonic(); }
  saveTo$rac$stack();
}
```

Figure 5.4: Evaluating old expressions appearing in constraints.

are evaluated in the pre-state, and their values are stored into fresh private fields. The constraint method refers to the private fields when evaluating the constraint predicate in the post-state.

Let $E = \{E_1, ..., E_k\}$ is be the set of old expressions appearing in the instance constraint, P. For each E_i, the JML compiler generates a new private field, `rac$old`$_i$, of type `JMLRacValue`. The class `JMLRacValue` is a wrapper class to encapsulate undefinedness (refer to Section 3.2 for undefinedness). The JML compiler also replace each E_i in P with the corresponding new field `rac$old`$_i$, i.e., $P' \equiv P[E_i := \texttt{rac\$old}_i]$, and translates the predicate, P', into a constraint method. Therefore, when evaluating the constraint predicate, the constraint method uses the new private fields in place of old expressions.

The new private fields, `rac$old`$_i$'s, must be initialized to the values of the corresponding old expressions, E_i's, evaluated in the pre-state. For this, the JML compiler generates a separate method, called an *old expression method* (see Figure 5.4), to be called by the wrapper method in the pre-state (see Section 5.4). The old expression method evaluates each old expression, E_i and stores the result to the corresponding private field, `rac$old`$_i$. If the evaluation of E_i results in undefinedness, one of two special values for undefinedness is stored to indicate this (see Section 3.2 for undefinedness). Thus, undefinedness is propagated from old expressions to the constraint predicate, and interpreted contextually. The meaning and need for the last statement, `saveTo$rac$stack()`, is explained in Section 5.3.2.

Why does the JML compiler generate separate old expression methods, instead of evaluating old expressions in precondition methods, e.g., as done for old expressions appearing in postconditions? The main reason is to support a modular way of inheriting specifications; this will be explained in detail in Section 6.4, but the essence is that a subtype's additional methods need a modular way to

```
public class Counter {
  private int cnt;
  //@ private constraint cnt >= \old(cnt);

  public void incr() { incr(1); }

  //@ requires x > 0;
  private void incr(int x) { cnt = cnt + x; }
}
```

Figure 5.5: Constraints and nested method calls.

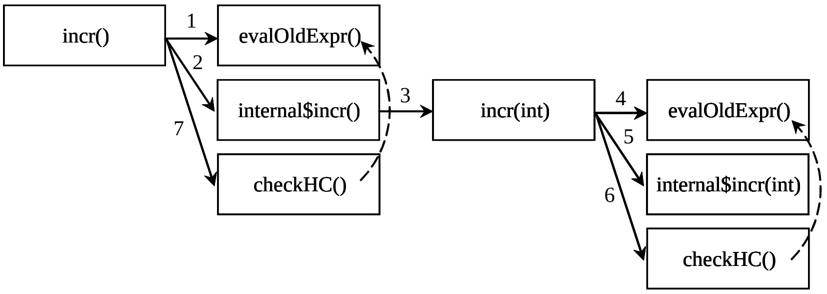

Figure 5.6: Referring to correct pre-state values in nested assertion checking.

evaluate the old expressions appearing in the inherited constraints.

The JML compiler handles old expressions appearing in static constraints in a similar way, except that the old expression method in this case becomes static.

5.3.2 Nested Method Calls

The old expressions appearing in constraints are evaluated in the pre-state by old expression methods, and their values are stored into new private fields (see Section 5.3.1). The constraint methods, called in the post-state, use these fields to refer to the pre-state values of the old expressions. However, there is a complication in this approach if methods calls other methods (including themselves) of the same type. The constraint methods need to refer old values evaluated in proper pre-states, as there are multiple pre-states during the course of method calls. Figure 5.5 shows a simple specification illustrating this complication, due to old expressions and nested assertion checking. The public method `incr()` calls the private method `incr(int)`. Thus, assertion checking becomes nested (see Figure 5.6). The old expression `\old(cnt)` appearing in the constraint is evaluated first in the pre-state of the method `incr()`, denoted by the method call 1, and then again in the pre-state of the method `incr(int)`, denoted by the method call 4. The constraint method is also called twice, first in the post-state of the method `incr(int)`, denoted by the method call 6, and second in the post-state of the method `incr()`, denoted by the method call 7. As illustrated by the dotted arrow, the complication is that the constraint method has to use proper pre-state values for the old expression `\old(cnt)`.

How does the runtime assertion checker correctly refer to pre-state values during nested assertion checking? It uses a private stack to save and restore pre-state values during nested assertion checking. The old expression methods store old values into the stack at the end, and the constraint methods restore them from the stack before evaluating constraints. The first is done by calling the method `saveTo$rac$stack` (see Figure 5.4) and the second by calling the method `restoreFrom$rac$stack`

```
private transient java.util.Stack rac$stack = new java.util.Stack();

private void saveTo$rac$stack() {
  java.lang.Object[] values = new java.lang.Object[1];
  values[0] = rac$old1;
  rac$stack.push(values);
}

private void restoreFrom$rac$stack() {
  java.lang.Object[] values = (java.lang.Object[]) rac$stack.pop();
  rac$old1 = (JMLRacValue) values[0];
}
```

Figure 5.7: Old value stacks generated by the JML compiler.

(see Figure 5.3).

The JML compiler, if necessary, automatically generates private stacks and their methods. Figure 5.7 shows an example of such stacks generated by the JML compiler for the class `Counter` in Figure 5.5. The method `saveTo$rac$stack` pushes all old values to the stack, i.e., just `rac$old1` in this example, and the method `restoreFrom$rac$stack` pops the stack to restore the old values for all private fields, i.e., just `rac$old1` in this example. Arrays are needed to handle the case where there is more than one old value.

A similar complication exists for method specifications if methods call themselves directly or indirectly. For each method, if necessary, the JML compiler generates a separate stack for storing and restoring pre-state values for the method, such as the precondition itself and old variables and expressions appearing in the postconditions. The pre- and postcondition methods call appropriate stack methods.

5.3.3 Method-specific Constraints

So far, this section considered only *universal constraints*, constraints that are applicable to all methods. In JML, it is possible to write a constraint that is applicable only to a certain set of methods; the names and signatures of applicable methods can be explicitly listed, or simple patterns can be specified. In this dissertation, this kind of constraints are called *method-specific constraints.* For example, the following is a method-specific constraint applicable only to two methods, `add(Object)` and `addAll(Collection)`; if the `for` clause is omitted, it becomes a universal constraint.

```
/*@ constraint contents.size() >= \old(contents.size())
  @   for add(Object), addAll(Collection); @*/
```

How does the runtime assertion checker support method-specific constraints? The JML compiler generates a separate constraint method for each method-specific constraint; a method-specific constraint method takes method names and signatures as arguments. Let $C_0, C_1, \ldots, C_n$ be instance constraints of a type, say T, where C_0 is a universal constraint, and $C_1, \ldots, C_n$ are method-specific constraints. If there are more than one universal constraint, they are assumed to be conjoined into C_0. Let $M_i = \{M_{i1}, \ldots, M_{im}\}$ be the constraining patterns (i.e., the `for` clause) for the constraint C_i, where $i = 1, \ldots, n$. Then, Figure 5.8 shows the extended instance constraint method, `checkHC$instance$`T, and newly-introduced, universal and method-specific constraint methods, `checkHC$instance`$_i$`$`T.

The instance constraint method is extended to take the name and signature of the method under runtime assertion checking. It first calls the universal constraint method, and then calls each

```
public void checkHC$instance$T(String forName, Class[] forSig) {
  restoreFrom$rac$stack();
  checkHC$instance_0$T();
  checkHC$instance_1$T(forName,forSig);
  ...
  checkHC$instance_n$TforName,forSig);
}

private void checkHC$instance_0$T() {
  ⟪checkC_0⟫
}

private void checkHC$instance_i$T(String forName, Class[] forSig) {
  ⟪testM_i⟫
  ⟪checkC_i⟫
}

⟪testM_i⟫ ≡
  boolean rac$v = false;
  if (!rac$v && (M_i1.contains(forName,forSig)) { rac$v = true; }
  ...
  if (!rac$v && (M_in.contains(forName,forSig)) { rac$v = true; }
  if (!rac$v) { return; }

⟪checkC_i⟫ ≡
  boolean rac$v;
  ⟦C_i, rac$v⟧
  if (!rac$v) { throw new JMLConstraintError(); }
```

Figure 5.8: Extended and new constraint methods for method-specific constraints.

method-specific constraint method. For the call to method-specific constraint methods, it passes as arguments the name and signature of the method being checked.

Each method-specific constraint method, `checkHC$instance`$_i$`$`T, $i = 1, \ldots, n$, first tests if the given method name and signature matches one of the method patterns, M_{ij}, specified in the `for` clause of the constraint, C_i. It then performs actual checking for the constraint C_i only if a match is found.

Wrapper methods are also extended to supply appropriate arguments (i.e., method names and signatures) to constraint method calls.

Static constraints methods are similarly extended to support static method-specific constraints Of course, static universal and and method-specific constraint methods, `checkHC$static`$_i$`$`T, become static.

5.4 Wrapper Methods Revisited

The assertion checking methods for type assertions, such as invariant and constraint methods, are expected to be called by wrapper methods. Figure 5.9 shows a revised structure of wrapper methods that takes into account of type assertions. The method m is assumed to be declared in a type named S. The wrapper method first checks invariants by calling invariant methods. The instance

```
T m(T1 x1, ..., Tn xn) {
  checkInv$static$S();
  [checkInv$instance$S();]
  checkPre$m$S(x1, ..., xn);
  evalOldExprInHC$static$S();
  [evalOldExprInHC$instance$S();]
  T rac$result;
  try {
    rac$result = orig$m(x1, ..., xn);
    checkPost$m$S(x1, ..., xn, rac$result);
    return rac$result;
  } catch (JMLEntryPreconditionError rac$e) {
    throw new JMLInternalPreconditionError(rac$e);
  } catch (JMLAssertionError rac$e) {
    throw rac$e;
  } catch (Throwable rac$e) {
    checkXPost$m$S(x1, ..., xn, rac$e);
  } finally {
    if (/* no postcondition violation? */) {
       checkInv$static$S();
       [checkInv$instance$S();]
       checkHC$static$S("m", /* T1, ..., Tn */);
       [checkHC$instance$S("m", /* T1, ..., Tn */);]
    }
  }
}
```

Figure 5.9: Revised structure of wrapper methods

invariant method, `checkInv$instance$`S, is called only if the method $m$ is an instance method; this is indicated by enclosing the statement inside a pair of square brackets ([·]). The wrapper method then checks the method's precondition. As the precondition may rely on invariants, the precondition is checked after the invariant is checked. The wrapper method next calls old expression methods to evaluate old expressions appearing constraints. The old expressions are evaluated last in the pre-state; there is no need to evaluate them if either the invariant or the precondition is violated. In the post-state, both invariants and constraints are checked after the method's postcondition is checked. If the postcondition is violated, type assertions are not checked in the post-state.

Wrappers for constructors are similarly extended to take into account of type assertions. In wrapper constructors, however, instance invariants are not checked in the pre-state, and instance constraints are not checked in the post-state. If a constructor throws an exception, instance invariants are not checked in the post-state either.

Wrappers for helper methods (or constructors) remains the same, as helper methods are exempted from the obligation of preserving type assertions.

5.5 Specifications for Interfaces

Assertion methods such as pre- and postcondition methods, invariant methods, and constraint methods are all added to *host classes*. For a class, the class itself is the host class. For an interface, however, the interface itself cannot be the host class because in Java an interface method must be abstract [55, Section 9.4]. For an interface, therefore, the JML compiler generates a separate

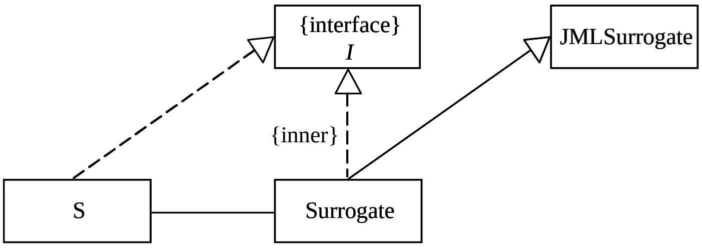

Figure 5.10: Interfaces and surrogate classes.

assertion class, called a *surrogate*, that hosts all assertion methods for the interface. That is, for an assertion specified in an interface, its host class is the surrogate class of the interface. Thus, the surrogate class is responsible for checking all the assertions specified in the interface. Another role of the surrogate class is to capture the specification state of the interface; an interface becomes stateful in JML, as it can have specification-purpose fields (see Section 6.5).

Let I be an interface with an implementing class S. The surrogate class of the interface I, named `Surrogate`, is declared as a static inner class of the interface I (see Figure 5.10). The surrogate class becomes a subclass of the class `JMLSurrogate`, a runtime class that defines properties common to all surrogate classes. The class `JMLSurrogate` also provides a facility for inheriting specifications through interfaces (see Section 6.5 for details). Each object of the class S is associated with a unique surrogate object of the interface I. The surrogate object is also responsible for representing the specification state, inherited from the interface I, of the implementing object.

The surrogate class I`.Surrogate` also implements the interface I. This is to facilitate evaluating assertions specified in the interface I. An assertion in the interface I may refer to methods that is declared in I but implemented in the implementing class S. This means that assertion methods for I must invoke methods defined in the class S. For this, the JML compiler uses the *delegation design pattern* [52]. Each surrogate object of I is associated with a unique instance of the class S. A method call to the surrogate object is delegated to the associated object of the class S. The surrogate object becomes the *delegate* and the associated object becomes the *delegated*.

Figure 5.11 shows an example delegation method to be defined in the surrogate class I`.Surrogate`. The delegation method, m, calls the corresponding method of the delegated, `self`; the field `self` is inherited from the superclass `JMLSurrogate`. It becomes interesting if such a method call encounters an exception. A distinction is made between instances of the class `java.lang.Error` and other exceptions. The first kind of exceptions are *angelic* in that they are contextually interpreted in such a way as to make the whole assertion true. This is because an implementation is freed from its obligation if it encounter a JVM error [89, Section 2.1.3.3] [128]. All other kinds of exceptions are *demonic* in that they are contextually interpreted in such a way as to falsify the whole assertion. Thus, the first kind of exceptions are translated into an angelic exception, and the rest are translated into a runtime exception, a standard demonic exception. In sum, exceptions encountered during delegation make the value of the method call undefined, and thus the method call is contextually interpreted by the parent expression (refer to Section 3.2.3 for details of contextual interpretation).

The surrogate class, if necessary, also defines delegation methods for the interface's implicitly declared abstract methods, such as `equals`, `hashCode`, `getClass`, `clone`, and `toString`. If an interface has no direct superinterfaces, the interface implicitly declares a public abstract method corresponding to each public instance method declared in the class `Object`, unless a method with the same signature is explicitly declared by the interface [55, Section 6.4.3].

```
public T0 m(T1 x1, ..., xn) {
  try {
    return ((I) self).m(x1, ..., xn);
  }
  catch (java.lang.Error rac$e) {
    throw new JMLAngelicException();
  }
  catch (java.lang.Throwable rac$e) {
    throw new java.lang.RuntimeException();
  }
}
```

Figure 5.11: Example of delegation methods to be defined in surrogate classes.

5.6 Discussion

5.6.1 Limitations

There is at least two limitations to the current approach of checking type invariants and constraints. One limitation is concerned with direction modification of public fields. Type assertions are checked only upon method calls, i.e., before and after method calls. They are not checked against state changes caused through other than method calls; e.g., clients may directly modify public fields. The JML semantics states that every client-visible state should preserve the properties specified by type assertions. One solution would be to inject assertion checking code directly into the client code for each reference of public fields [53] [133]; e.g., an assignment may be replaced with a call to the setter method of the field that also checks type assertions at the end. However, the approach would not be modular as it requires whole program analysis; i.e., the client code need be instrumented.

Another limitation of the current approach is checking type assertions for constructors. A static invariant should be preserved by a non-helper constructor of a class, and an instance invariants should be established by a non-helper constructor. Thus, the instance invariant is not checked in the pre-state of a constructor. An instance invariant is irrelevant to the pre-state of a constructor because there is no well-defined pre-state for checking the instance invariant. However, a static invariant should be checked in the pre-state of a non-helper constructor. This poses a problem because, as discussed in the previous chapter, an explicit constructor invocation statement, if such a call exists, should be the first statement of a constructor body [55, Section 8.8.5.1]. In the current approach, the static invariant is checked after the explicit constructor call if exists. A possible fix would be to generate Java bytecode directly without using intermediate instrumented source code as done by Lackner et al [82]. As Java bytecode does not enforce the above restriction, a call to an assertion check method can be inserted into the correction position of a constructor body.

5.6.2 Establishing Static Invariants

JML states that static invariants should be established by the initialization of a class, and they should be preserved by all non-helper constructors and methods. The question is how the runtime assertion checker checks whether static invariants are established by the initialization process. According to the Java Language Specification [55, Section 12.4], the initialization of a class consists of executing its static initializers and the initializers for static fields declared in the class. (For an interface, the initialization consists of executing the initializers for fields declared in the interface.) In addition, the static initializers and class variable initializers are executed in the textual order. The approach that the JML compiler takes is to add a special static initializer, called a *static invariant initializer*, at the end of a class declaration. The static invariant initializer calls the static invariant method of the class, and has the following form.

```
static { checkInv$static$T(); }
```

As the static initializers and class variable initializers are executed in the order of declarations, the added static invariant initializer is executed after all static fields complete their initializations, and thus static invariants are checked in the correct state.

5.6.3 Privacy of Specifications

It is possible to write type assertions of different visibility levels. An interesting question is then what the relationship should be among type assertions of different visibility levels. For example, should the public invariant imply the protected invariant? If one views the invariants as being conjoined to the method's pre- and postconditions and the constraints being conjoined to the method's postcondition, one comes up with the following properties, which are called *modular reasoning requirements.*

$$I^i_{\text{pub}} \Rightarrow I^i_{\text{pro}} \Rightarrow I^i_{\text{pkg}} \Rightarrow I^i_{\text{pri}} \tag{5.1}$$

$$I^s_{\text{pub}} \Rightarrow I^s_{\text{pro}} \Rightarrow I^s_{\text{pkg}} \Rightarrow I^s_{\text{pri}} \tag{5.2}$$

$$I^i_{\text{pub}} \wedge I^i_{\text{pro}} \Rightarrow I^i_{\text{pkg}} \wedge I^i_{\text{pri}} \tag{5.3}$$

$$C^i_{\text{pub}} \wedge C^i_{\text{pro}} \Rightarrow C^i_{\text{pkg}} \wedge C^i_{\text{pri}} \tag{5.4}$$

The notation I^i_{v} denotes the instance invariant of the privacy level v, I^s_{v} denotes the static invariant of the privacy level v, and C^i_{v} denotes the instance constraint of the privacy level v. If there are more than one type assertion at the same privacy level, they are assumed to be conjoined by desugaring. The first two properties are consequences of treating invariants as being conjoined to the method's preconditions. A public client may call a public method in a state where the public invariant is satisfied. The client would be surprised should he encounter a pre-state invariant violation caused by the invariants of other privacy levels; the only invariant visible to him is the public invariant. Therefore, the public invariant should imply the invariants of other privacy levels. For the similar reason, we have $I_j \Rightarrow I_k$, where I_j is more widely visible than I_k.

The properties (5.3) and (5.4) are consequences of both method overriding and treating invariants and constraints as being conjoined to the method's postconditions. Thus, they are not pertinent to `final` classes. A subclass may override a method inherited from its superclass. Since the subclass can only see the public and protected type assertions of the superclass, the subclass's overriding method will be able to establish only the supertype's public and protected type assertions. This causes a problem in reasoning about the supertype's code. When reasoning about the supertype's code, one can use type assertions of all privacy levels available, in particular, package-visible and private assertions. If the overriding method does not establish the supertype's type assertions, in particular, package-visible and private assertions, then reasoning about the supertype's code would be broken. Under the properties (5.3) and (5.4), however, the reasoning would still hold even when the subtype is added to the program. As static assertions are not inherited by subtypes, they are not constrained by the modular verification requirements. In sum, the properties support modular reasoning of code in the presence of subtyping [84] [91].

Chapter 6

Inheritance of Specifications

In JML, specification inheritance ensures that the behavior of a supertype is imposed on its subtypes, thereby supporting behavioral subtyping [40]. Furthermore, a subtype may inherit specifications from more than one supertype, thus allowing multiple inheritance.

In this chapter, I explain how the JML compiler supports inheritance of specifications. The motivating goal is to support separate compilation, the ability to compile source files separately. I introduce a delegation approach as a solution. In the delegation approach, a subtype's assertion methods call the corresponding assertion methods of its supertypes. The calls are made dynamically using Java's reflection facility. The various assertion methods, introduced in the previous chapters, are extended to inherit such specifications as pre- and postconditions, invariants, and constraints. In addition, several new assertion methods are introduced to support weak behavioral subtyping. Finally, I discuss how the statefulness of interface specifications affects the delegation approach.

6.1 Introduction

In JML, a subtype inherits specifications from its supertypes. A class inherits specifications from its superclasses and the interfaces that it implements. An interface inherits specifications from its superinterfaces. As in Java, only public and protected specifications are inherited; package-visible and private specifications are not inherited by subtypes[1].

What kinds of specifications are inherited? All kinds of specifications are inherited, including method specifications, invariants, constraints, specification-only declarations, `represents` clauses, `depends` clauses, and annotations such as `non_null` and `pure`. As in Java, however, only instance specifications are inherited; static specifications are not inherited.

What is the semantics of inheritance? JML imposes the specifications of supertypes on subtypes in order to achieve *behavioral subtyping* [40]. Inherited type assertions such as invariants and constraints can be thought of being conjoined to inheriting assertions in that a subtype has to satisfy both the inheriting and inherited type assertions. Method specifications are also inherited in such a way to guarantee behavioral subtyping. They are conjoined; preconditions are disjoined and postconditions are conjoined in the form of $\bigwedge(\texttt{\textbackslash old(}P_i\texttt{)} \Rightarrow Q_i)$, where P_i is a precondition and Q_i is the corresponding postcondition [130]. JML also supports *weak behavioral subtyping* in which a subtype's additional methods are relieved from satisfying history constraints inherited from its supertypes [38] [39] [40]. An additional method is a subtype's method that does not override any methods of its supertypes. Names appearing in (inherited) assertions are resolved in the same way as in Java; field names are statically resolved, and static method are statically dispatched whereas instance method are dynamically dispatched.

[1] In refinement, however, specifications of all privacy levels are inherited [89].

6.1.1 Challenges

Two main challenges in supporting specification inheritance are separate compilation and multiple inheritance. The JML compiler should be able to compiler source files separately. However, separate compilation means that, when compiling a subtype, the compiler does not know whether the supertypes of the type being compiled will be compiled with runtime assertion check or not. The compiled bytecode of the subtype should work regardless of the compilation choice of its supertypes.

JML supports multiple inheritance of specifications, as specifications are inherited from both superclasses and (super)interfaces [89]. As Java supports only single inheritance of code, the JML compiler cannot completely rely on Java's inheritance mechanism to implement inheritance of specifications in JML.

Needless to say, the JML compiler should be faithful to the semantics of JML for specification inheritance, e.g., resolving names, static versus dynamic dispatch, and composition of various assertions.

6.1.2 Approach

There are at least two approaches to supporting specification inheritance.

- *Textual copy approach*: A supertype's specifications are textually copied down to a subtype, and are translated into runtime assertion checking code as parts of the subtype's specifications [13] [25] [85] Often, desugaring (and renaming) is needed to combine inheriting and inherited specifications [130].

- *Delegation approach*: A subtype delegates to its supertypes the responsibility of checking inherited specifications, e.g., by calling appropriate assertion checking methods of the supertypes [28]. To support separate compilation, delegation should be done dynamically, as the supertypes may not contain assertion checking code; the supertypes may not be compiled for runtime assertion checking.

The textual copy approach is more efficient than the delegation approach, because it incurs no runtime overhead for delegation. However, the approach is not modular in that it needs supertypes' specifications to compile subtypes. Another problem is that some names appearing in the inherited specifications must be resolved statically in the scope of the supertypes. Examples of such names are fields, types, and static methods. Resolving such names is not easy, especially, in the presence of name hiding and `super` calls [55, Chapter 6].

The delegation approach is modular; supertypes' specifications are not needed to compile subtypes. A shortcoming of the approach is the runtime performance due to (dynamic) delegation. However, dynamic nature is not inherent in the delegation approach and static form of delegation is also possible (see Section 9.1). The JML compiler uses the delegation approach.

6.1.3 Delegation Approach

In the delegation approach, dynamic calls are used to inherit specifications from supertypes. The term *dynamic call* refers to a method call made by using Java's reflection facility [146]. The underlying idea of the delegation approach is for a subtype's assertion methods such as pre- and postcondition methods, invariant methods, and constraint methods to make dynamic calls to the corresponding assertion methods of its supertypes. The reason for using dynamic calls is that, due to separate compilation, a subtype does not know whether its supertypes will have the corresponding assertion methods. Even though assertion methods are dynamically called, their names and types are statically determined at compile-time, since the call site contains this information.

In general, each assertion method performs the followings.

1. Check assertions specified in the current type.
2. Check the inherited assertions. For each statically-determined pair of type and method names, perform the following steps.
   (a) Look up the corresponding assertion method in the supertype.
   (b) Invoke the assertion check method, if found, on the calling object (`this`). This implements a dynamic form of delegation, as it supports *down calls*, calls from the supertype to the subtype's overriding methods.
   (c) Invoke the assertion method found in the supertype.
   (d) Combine the results, e.g., disjunction for preconditions and conjunction for invariants.
3. Report an assertion violation if the combined result does not hold.

Several optimization techniques are used to minimize the performance overhead caused by the dynamic approach. For example, the pre- and postcondition methods of a subtype's additional methods do not attempt to make dynamic calls to its supertypes; a subtype's additional methods do not inherit method specifications form its supertypes. It was observed that dynamic invocation itself is only marginally worse than static invocation, but introspection, i.e., looking up types and methods, is costly. Thus, local caches are used to reduce the number of dynamic lookups.

The rest of this chapter discusses how the delegation approach affects and extends different kinds of assertion methods, such as pre- and postcondition methods, invariant methods, and constraint methods, to support specification inheritance. The extension described in this chapter applies only to instance assertion methods; because static specifications are not inherited by subtypes, no such extension is needed for static assertion methods.

6.2 Method Specifications

In this section, assertion methods for method specifications defined in Chapter 4 are extended to support inheritance of specifications. In addition to multiple inheritance, several things complicate inheritance of method specifications.

- *Pre-state expressions*: In addition to locally specified pre-state expressions such as old variables, old expressions, and preconditions themselves (see Section 4.4), all inherited pre-state expressions must be evaluated in the pre-state, and their values must be available to the corresponding, inherited postconditions.
- *Classes versus interfaces*: Method specifications can be inherited both from superclasses and (super)interfaces. As assertion methods for interfaces are hosted by surrogate classes (see Section 5.5), dynamic call mechanism needs to be adjusted to inherit specifications from interfaces.
- *Dynamic Semantics*: Java does dynamic dispatch for dynamic calls. If the assertion methods of a subclass and its superclass have the same name, a dynamic call to the assertion method of the superclass upon an instance of the subclass ends up calling the assertion method of the subclass. To prevent this, type names are appended to the names of assertion methods, e.g., `checkPre$m$T`, where `T` is the type name.

In the following, I first introduce the notion of immediate supertypes for specification inheritance and then explain each assertion method such as pre- and postcondition methods extended for specification inheritance.

6.2.1 Immediate Supertypes for Specification Inheritance

Instead of attempting dynamic calls to all direct and indirect supertypes for specification inheritance, the runtime assertion checker makes dynamic calls only to a set of statically-determined supertypes. Such a set of supertypes is determined at compile-time based on method overriding and interface implementation. If a method does not override a superclass' method, the method cannot inherit any method specifications from the superclass. Similarly, a method cannot inherit any method specifications from an interface unless it provides an implementation to an abstract method declared in the interface. Further, it is sufficient to call the corresponding assertion method of the nearest superclass (or superinterface) whose method is overridden (or implemented) by the method under assertion checking; such an assertion method will call its nearest assertion method, and so on. For this, I formulate the notion of immediate supertypes for specification inheritance. Let m be an instance method declared in a type S. A type T is an *immediate supertype of the type S for the specification inheritance with respect to the method* m if either of the following conditions is satisfied.

1. If T is a class, then
   (a) T is a direct or indirect superclass of the class S,
   (b) T declares a method m that is overridden by the method m of S (refer to [55, Section 8.4.6] for Java's method overriding rules), and
   (c) there is no intermediate class between the class S and the class T that overrides the method m of T. An intermediate class is a class that is a proper superclass of the class S and a proper subclass of the class T.
2. If T is an interface, then
   (a) S implements (or extends) T directly or indirectly,
   (b) T declares a method m that is overridden by the method m of S (refer to [55, Section 8.4.6 and 9.4.1] for Java's method overriding rules), and
   (c) there is no intermediate interface between S and T that overrides the method m of T. An intermediate interface is a proper subinterface of T that is implemented (or extended) by S, and
   (d) T is not an immediate supertype of any classes determined by the condition 1 above, if such classes exist.

For each inheritance chain, an immediate supertype for that chain is the nearest supertype that also declares the method under consideration. The condition 2c prevents some inheritance chains from being considered for the immediate superclass for specification inheritance. For example, a direct superinterface of a class S is not S's immediate supertype for specification inheritance if the interface is an immediate supertype for S's superclasses. However, the above definition does not eliminate such duplications between two interface inheritance chains. I also use the terms an *immediate superclass for specification inheritance* and an *immediate superinterface for specification inheritance* when the supertype is a class and an interface respectively. If S is a class, it can have at most one immediate superclass but more than one immediate superinterface for specification inheritance, because Java permits only a single superclass, but multiple interfaces. If S is an interface, it can have no immediate superclass, but can have multiple immediate superinterfaces for specification inheritance.

Let Σ be the set of immediate supertypes for specification inheritance with respect to a method m, declared in a type S. An assertion method for the method m in the type S calls dynamically the corresponding assertion method of each type in the set Σ (see Section 6.2.2, 6.2.3, and 6.2.4). It is clear that the method m inherits all method specifications applicable to it, both from class specifications and interface specifications.

6.2.2 Preconditions

A method should be called in a state where its precondition — explicitly specified or inherited from its supertypes — holds. In JML, the semantics is to disjoin the inherited preconditions with the local, explicitly specified one [130]. Let P_0 be a method's explicitly specified precondition, and let $P_1, \ldots, P_n$ be inherited preconditions. Then, the method's *effective* precondition, P, is $P_0 \vee P_1 \vee \ldots \vee P_n$. Remember that each precondition P_i is checked by the precondition method added to the owner class of P_i (see Section 4.4).

Let m be an instance method declared in this type, S, with the following header, where the access modifier V is either `public` or `protected`. (Except for refinement, only `public` or `protected` methods inherit method specifications from supertypes.)

$V\ T\ m(T_1\ x_1,\ \ldots,\ T_n\ x_n)$ `throws` $E_1,\ \ldots,\ E_m$

Suppose that the method m has a precondition P_0 specified in the type S. Let $I_1, ..., I_k$ be immediate supertypes of S for specification inheritance with respect to the method m. If S is a class, then at most one of I_i's can be a class and the rest are interfaces; otherwise, all I_i's are interfaces. Figure 6.1 shows the precondition method of m in the type S extended for specification inheritance (refer to Section 4.4 for the original definition). The extended method first evaluates the local precondition P_0 to set P_0's precondition field, `rac$pre`$_0$. A *precondition field* is a private field added by the JML compiler (see Section 4.6), and lets the postcondition method refer to the value of the corresponding precondition (see Section 6.2.3 and 6.2.4). The precondition field `rac$pre`$_0$ becomes the initial value of the local variable `rac$`b, denoting the effective precondition of the method. Each code chunk ⟪*checkPrei*⟫ calls dynamically the corresponding precondition method of type I_i to set the precondition field `rac$pre`$_i$ and disjoin it to the effective precondition `rac$`b (see below). Finally, if the effective precondition `rac$`b does not hold, the method throws a precondition violation error.

Each code chunk ⟪*checkPre*$_i$⟫$\equiv$, for $i = 1, \ldots, k$, inherits the corresponding precondition from the immediate supertype I_i. It calls dynamically the precondition method `checkPre$`m`$`I_i of type I_i on the receiver `this`. The utility method `rac$check` makes such a dynamic call by using Java's reflection facility, and masks all exceptions except for assertion violation errors. For example, it finds in the type I_i a method named `checkPre$`m`$`I_i of signature $T_1, ..., T_n$ and, if found, invokes the method on the receiver `this` with the arguments $x_1, ..., x_n$; the details of argument passing for dynamic calls are suppressed. Depending on the result of dynamic call, the precondition field `rac$pre`$_i$ is set to either true of false; it becomes true only if the method `checkPre$`m`$`I_i exists in I_i and the method call does not results in an assertion violation error (i.e., a precondition violation). The precondition field `rac$pre`$_i$ is also disjoined to the effective precondition `rac$`b. One thing to note is that if type I_i has no corresponding precondition method, it does not contribute to the effective precondition, `rac$`b. Thus, one consequence of the delegation approach (and separate compilation) is that the effective precondition of the method under assertion checking becomes the disjunction of all inherited preconditions that are compiled with runtime assertion checking.

It can be proved by induction that the extended precondition method checks the predicate, $\bigvee P_i$, where P_0 be the local precondition and other P_i's are inherited ones. The precondition method also evaluates all pre-state expressions for use in the evaluation of postconditions; this aspect is suppressed in the code above (refer to Section 4.6 for details).

6.2.3 Normal Postconditions

The postcondition methods are extended in a similar fashion to support inheritance of postconditions. However, there are two important differences: (1) inherited postconditions are conjoined, and (2) postconditions need hold only when the corresponding preconditions hold. As before, let m be an

```
public void checkPre$m$S(T_1 x_1, ..., T_n x_n) {
  ⟦P_0, rac$pre_0⟧
  boolean rac$b = rac$pre_0;
  ⟪checkPre_1⟫
  ...
  ⟪checkPre_k⟫
  if (!rac$b) {
    throw new JMLEntryPreconditionError();
  }
}

⟪checkPre_i⟫ ≡
  rac$pre_i = false;
  try {
    rac$check("I_i", this, "checkPre$m$I_i", /* T_1 x_1, ..., T_n x_n */);
    if ("checkPre$m$I_i" exists in I_i?) {
       rac$pre_i = true;
       rac$b = true; // i.e.,  rac$b = rac$b || rac$pre_i;
    }
  }
  catch (JMLAssertionError rac$e) {
  }
```

Figure 6.1: Extended precondition methods.

instance method declared in a type, S, with the following header:

$V\ T\ m$(`$T_1$ $x_1$, ..., $T_n$ $x_n$`) `throws` E_1, ..., E_m

Let Q_0 be the normal postcondition specified in the type S, and $I_1, ..., I_k$ be immediate supertypes of S for specification inheritance with respect to the method $m$. Then, the normal postcondition method for $m$ in the type S is extended as shown in Figure 6.2 (refer to Section 4.5.1 for the original definition). The extended method first checks the local postcondition, Q_0, if the corresponding precondition, say P_0, holds. Remember that the precondition, P_0, is evaluated in the pre-state and is stored into the precondition field, `rac$pre`$_0$, by the precondition method (see Section 6.2.2). The method then checks each inherited postcondition in the code chunk $\langle\langle$*checkPost*$_i\rangle\rangle$, where $i = 1, \ldots, k$. The inherited postconditions are checked only if both the corresponding preconditions `rac$pre`$_i$'s and the conjunctively-accumulated, effective postcondition `rac$`b hold. The `catch` clause identifies the violation of inherited postconditions, as the method `checkPost$`m`$`I_i would throw a postcondition violation error if the postcondition of $m$ inherited from I_i gets violated. Finally, the method throws a postcondition violation error if the effective postcondition does not hold.

In sum, the behavior of the extended normal postcondition method is to check in a short-circuit form the predicate, $\bigwedge(\texttt{\textbackslash old}(P_i) \Rightarrow Q_i)$ where P_0 and Q_0 are local pre- and normal postconditions and other P_i's and Q_i's are inherited pre- and normal postconditions.

6.2.4 Exceptional Postconditions

The treatment of exceptional postcondition methods is similar to that of normal postcondition methods. An exceptional postcondition method, however, has to make assertion checking transparent by rethrowing the original exception thrown by the method under assertion checking. Let $m$ be an

```
public void checkPost$m$S(T_1 x_1, ..., T_n x_n, T rac$result) {
  boolean rac$b = true;
  if (rac$pre_0) {
    ⟦Q_0, rac$b⟧
  }
  ⟪checkPost_1⟫
  ...
  ⟪checkPost_k⟫
  if (!rac$b) {
    throw new JMLNormalPostconditionError();
  }
}

⟪checkPost_i⟫ ≡
  if (rac$b && rac$pre_i) {
    try {
      rac$check("I_i", this, "checkPost$m$I_i",
                /* T_1 x_1, ..., T_n x_n, T rac$result */);
    }
    catch (JMLAssertionError rac$e) {
      rac$b = false;  // postconditions inherited from I_i violated
  }
```

Figure 6.2: Extended normal postcondition methods.

instance method declared in a type, S, with the following header:

$V\ T\ m(T_1\ x_1,\ \ldots,\ T_n\ x_n)$ `throws` $E_1,\ \ldots,\ E_m$

Let R_0 be the exceptional postcondition specified in the type S, and $I_1, ..., I_k$ be immediate supertypes of S for specification inheritance with respect to the method $m$. Then, the exceptional postcondition method for $m$ in the type S is extended as shown in Figure 6.3 (refer to Section 4.5.2 for the original definition). As in the normal postcondition method, the local exceptional postcondition, R_0, is first checked, but only if the corresponding precondition holds. Checking R_0 consists of case analysis on the thrown exception, `rac$thrown`; the details are suppressed (see Section 4.5.2 for details). Then, each inherited exceptional postcondition is checked. The mechanism is the same as that of the normal postcondition method. If the effective exceptional postcondition does not hold, the method throws an exceptional postcondition violation error. Finally, the method rethrows the original exception by performing a case analysis (see Section 4.5.2 for details).

In sum, the extended exceptional postcondition method checks a short-circuit form the predicate, $\bigwedge$(`\old(`P_i`)` $\Rightarrow R_i$), where P_0 and R_0 are the local pre- and exceptional postconditions, and the rest of P_i's and R_i's are inherited pre- and exceptional postconditions.

6.3 Invariants

For the inheritance of Instance invariants, the instance invariant method is extended to call the instance invariant methods of its direct supertypes. Let S be a type with the desugared instance invariant, I_0, and direct supertypes, $T_1, \ldots, T_n$. If S is a class, then it can have at most one direct superclass, and the rest are direct interfaces that it implements; otherwise, each T_i is a superinterface that the interface S directly extends. Then, the instance invariant method of type S is extended as

```
public void checkXPost$m$S(T1 x1, ..., Tn xn, Throwable rac$thrown) {
  boolean rac$b = true;
  if (rac$pre0) {
     ⟦Q0, rac$b⟧
  }
  ⟪checkXPost1⟫
  ...
  ⟪checkXPostk⟫
  if (!rac$b) {
    throw new JMLExceptionalPostconditionError();
  }
  ⟪rethrowException⟫
}

⟪checkXPosti⟫ ≡
  if (rac$b && rac$prei) {
    try {
      rac$check("Ii", this, "checkXPost$m$Ii",
                /* T1 x1, ..., Tn xn, Throwable rac$thrown */);
    }
    catch (JMLAssertionError rac$e) {
      rac$b = false;
  }
```

Figure 6.3: Extended exceptional postcondition methods. The code chunk ⟪*rethrowException*⟫ is defined in Section 4.5.2.

shown in Figure 6.4. The extended method first checks the instance invariant, I_0, explicitly specified in the type S, and then it dynamically calls the invariant method of each supertype, T_i, with the result being conjoined (see the code chunk ⟪*checkInvT$_i$*⟫). If the conjoined, effective invariant does not hold, the method throws an invariant violation error. The code chunk ⟪*checkInvT$_i$*⟫ ≡ dynamically calls to the instance invariant method of the type T_i. If the dynamic call encounters an assertion violation error, the effective invariant, rac$*b*, becomes false; such an assertion violation error indicates that the inherited invariant is violated. The inherited invariants are short-circuit evaluated, as the dynamic call is made only if the value of rac$*b* is true; the motivation is to stop as soon as the runtime assertion checker finds an assertion violation error.

In sum, the extended instance invariant method checks the invariants of a type, explicitly specified or inherited. The explicitly specified invariants are directly evaluated by the invariant method while the inherited ones are checked indirectly by dynamically calling the invariant methods of direct supertypes.

6.4 Constraints

For the inheritance of instance constraints, the instance constraint method is extended in a similar way as in the instance invariant method (see Section 6.3). Let S be a type with a local instance constraint, C_0, and direct supertypes, $T_1, \ldots, T_n$. Then, the instance constraint method of type S is extended as shown in Figure 6.5. The extended method first evaluates the local instance constraint C_0. Then, it checks the inherited constraints by dynamically calling the instance constraint methods of direct supertypes $T_1, \ldots, T_n$. The results are conjoined, and if the conjoined, effective constraint does not hold, the method throws an constraint violation error.

```
public void checkInv$instance$S() {
  boolean rac$b = true;
  ⟦I_0, rac$b⟧
  }
  ⟪checkInvT_1⟫
  ...
  ⟪checkInvT_n⟫
  if (!rac$b) {
    throw new JMLInvariantError();
  }
}

⟪checkInvT_i⟫≡
  if (rac$b) {
    try {
      rac$check("T_i", this, "checkInv$instance$T_i", /* ... */);
    }
    catch (JMLAssertionError rac$e) {
      rac$b = false;
  }
```

Figure 6.4: Extended invariant methods.

```
public void checkHC$instance$T(String forName, Class[] forSig) {
  restoreFrom$rac$stack();
  boolean rac$b = true;
  ⟦C_0, rac$b⟧
  }
  ⟪checkHCT_1⟫
  ...
  ⟪checkHCT_n⟫
  if (!rac$b) {
    throw new JMLConstraintError();
  }
}

⟪checkHCT_i⟫≡
  if (rac$b) {
    try {
      rac$check("T_i", this, "checkHC$instance$T_i", /* forName, forSig */);
    }
    catch (JMLAssertionError rac$e) {
      rac$b = false;
  }
```

Figure 6.5: Extended constraint methods.

```
public void evalOldExprInHC$instance$S() {
  ⟦E_1, rac$old_1⟧
  ...
  ⟦E_m, rac$old_m⟧
  saveTo$rac$stack();
  rac$check("T_1", this, "evalOldExprInHC$instance$T_1", /* ... */);
  ...
  rac$check("T_n", this, "evalOldExprInHC$instance$T_n", /* ... */);
}
```

Figure 6.6: Extended old expression methods.

A history constraint may contain old expressions to relate the current state to the earlier states. In runtime assertion checking, this means that all such old expressions including the inherited ones must be evaluated in the pre-state so that their values become available to the constraint methods. For this, extended is the old expression method introduced in Section 5.3.1.

Let $E_1, \ldots, E_m$ be old expressions appearing in C_0. Then, the extended old expression method has the structure shown in Figure 6.6. The extended method first evaluates old expressions appearing in C_0, $E_1, \ldots, E_m$, storing the results into old expressions fields, `rac$old`$_1$, . . . , `rac$old`$_m$ respectively. If the evaluations encounter exceptions, special undefinedness values are stored into the old expression fields (see Section 5.3.1 for details). As before, the old expression fields are saved into a stack to allow nested or recursive method methods. (see Section 5.3.2). The method then dynamically calls the instance old expression methods of each direct supertype, T_i, thus evaluating old expressions of T_1.

6.4.1 Weak Behavioral Subtyping

A subtype object has to preserve the instance constraints specified by its supertypes. In particular, the subtype's *additional methods* — new methods of the subtype that do not override inherited ones — have to establish in the post-state its supertypes's instance constraints. This stringent requirement is called *strong behavioral subtyping* [101]. If the subtype's additional methods are relieved from obeying the type constraints inherited from its supertypes, it is called *weak behavioral subtyping* [38] [39] [40]. In weak behavioral subtyping, however, the subtype's overriding methods still have to preserve the inherited constraints. In sum, a method m of type S has to satisfy all the following instance constraints.

1. from S, both universal constraints and method-specific constraints applicable to m,
2. from each supertype, T, of S,
   (a) if S is a strong subtype of T, both universal constraints and method-specific constraints applicable to m,
   (b) if S is a weak subtype of T,
      i. if m is an overriding method, both universal constraints and method-specific constraints applicable to m,
      ii. otherwise, nothing.

In JML, the `weakly` annotation specifies weak behavioral subtyping. If omitted, it defaults to strong behavioral subtyping. Figure 6.7 shows an example specification of weak behavioral subtyping. The class S is a strong behavioral subtype of the class T, but a weak behavioral subtype of the interface I.

```
public interface I { /* ... */ }
public class T { /* ... */ }
public class S extends T implements I /*@ weakly @*/ { /* ... */ }
```

Figure 6.7: Example specification of weak behavioral subtyping.

The notion of weak behavioral subtyping, in conjunction with method-specific constraints, poses a challenge to the runtime assertion checker. Given a method, the runtime assertion checker has to determine whether the method overrides any inherited methods or not. The decision should be made relative to the type whose constraints are being inherited; this should be done at runtime to support separate compilation.

The JML compiler's approach to addressing this challenge is, in addition to existing constraint methods such as local constraint methods and mother constraint methods (see Section 5.3.3), to introduce two new constraint methods corresponding to two forms of behavioral subtyping: a *weak constraint method* and a *strong constraint method.* They are for checking constraints on behalf of the subtype's methods, and thus called *subtyping constraint methods*, and The weak constraint method will be called by weak subtypes, and the strong one by strong subtypes. A subtyping constraint method checks constraints applicable to the method whose name and signature are given as arguments; i.e., method names and signatures are passed as arguments to the subtyping constraint method. As before, a universal constraint is applicable to every method and a method-specific constraint is applicable to a particular set of methods (see Section 5.3.3). However, the weak constraint method checks applicable constraints only if the given method is also declared in the current (owner) type or its supertypes, while the strong constraint method checks them regardless of method overriding. That is, the weak constraint method enforces constraints provided that the argument method is an overriding one. The decision about method overriding is made dynamically by using Java's reflection facility. Since only instance constraints are inherited by subtypes, the subtyping constraint methods are generated only for instance constraints.

In addition to subtyping constraint methods, the JML compiler still generate the usual constraint methods such as local constraint methods and mother constraint methods (see Section 5.3.3). As before, the mother constraint method checks locally-specified constraints by calling local constraint methods, and then it inherits supertypes's constraints, by calling their subtype constraint methods, either the strong ones or the weak ones. If the current (owner) type is a strong subtype, it calls the strong subtype constraint method of the supertype; otherwise, it calls the weak one.

Figure 6.8 shows a new set of constraint methods to support both strong and weak behavioral subtyping. The difference from the constraint methods of Section 5.3.3 is the introduction of subtyping constraint methods such as `checkHC$instanceS$`S for strong subtypes and `checkHC$instanceW$`S for weak subtypes, and rewiring to appropriate subtyping constraint methods to inherit constraints from supertypes. In the mother constraint method, `checkHC$instance$`S, the argument method's name (`forName`) is set to null if the method is private (including package-visibility) prior to calling constraint methods of supertypes. This special treatment is to prevent the subtype's private methods from inheriting the supertypes' method-specific constraints applicable to the supertype's private methods (which may have the same names and signatures as those of the subtype). In the code chunk ⟨⟨*checkInheritedHC*⟩⟩ ≡, calls to subtyping constraint methods are made dynamically using Java's reflection facility; however, this detail is suppressed. The static method `JMLChecker.isInheritedFrom`, called by the weak subtype constraint method, tests if the given method is inherited from the given type or its supertypes; this is decided at runtime by using Java's reflection facility.

```
/** Mother constraint method for type S. */
public void checkHC$instance$S(boolean isPri, String forName,
                               Class[] forSig) {
  restoreFrom$rac$stack();
  ⟪checkLocalHC⟫
  forName = isPri ? null : forName;
  ⟪checkInheritedHC⟫
}

⟪checkLocalHC⟫≡ /* universal and method-specific methods */
  checkHC$instance_0$S(forName,forSig);
  checkHC$instance_1$S(forName,forSig);
  ...
  checkHC$instance_m$S(forName,forSig);

⟪checkInheritedHC⟫≡ /* weak and strong subtyping methods */
  checkHC$instance[W|S]$T_1(forName,forSig);
  ...
  checkHC$instance[W|S]$T_n(forName,forSig);

/** Strong constraint method for type S */
public void checkHC$instanceS$S(String forName, Class[] forSig) {
  restoreFrom$rac$stack();
  ⟪checkLocalHC⟫
  ⟪checkInheritedHC⟫
}

/** Weak constraint method for type S */
public void checkHC$instanceW$S(String forName, Class[] forSig) {
  restoreFrom$rac$stack();
  if (JMLChecker.inheritedFrom(S.class,forName,forSig)) {
    ⟪checkLocalHC⟫
  } else {
    forName = null;
  }
  ⟪checkInheritedHC⟫
}
```

Figure 6.8: Constraint methods to support both strong and weak behavioral subtyping.

6.5 Specifications for Interfaces

A surrogate for an interface may have state of its own, due to model and ghost fields declared in the interface. This means that the runtime assertion checker cannot create a new surrogate object for an interface each time it needs to inherit specifications from the interface. A surrogate class is a separate assertion class hosting all the assertion methods of an interface (see Section 5.5). As a static inner class of the interface, it is responsible for checking assertions specified in the interface. A subtype of an interface inherits the interface's specifications by dynamically calling the assertion methods defined in the interface's surrogate class. For example, when a subtype's precondition method dynamically calls the corresponding precondition methods of its immediate supertypes, the

```
public interface Countable {
  //@ instance ghost int count initially count == 0;

  /*@ assignable count;
    @ ensures count == \old(count + 1); */
  void increment();

  //@ ensures \result == count;
  int value();
}

public class Counter implements Countable {
  private int cnt = 0;
  public Counter() { /*@ set count = 0; @*/ }
  public void increment() { cnt++; /*@ set count = cnt; @*/ }
  public int value() { return cnt; }
}
```

Figure 6.9: Example of stateful interfaces.

call is directed to the surrogate class if the immediate supertype is an interface. This kind of assertion method calls is termed as a *dynamic assertion call.* When an interface assertion refers to a method declared in the interface, called is the corresponding method of the implementing class. This kind of method calls is termed as a *static delegation call.* The object that receives such delegation calls from a surrogate object, is referred to as the *delegee* of the surrogate object.

The statefulness of surrogates complicates the inheritance of specifications from interfaces. Figure 6.9 shows an example of an interface with its own state. The interface `Countable` declares a ghost field `count`, whose abstract value may be changed. As an example, consider the following piece of code.

```
Counter ctr = new Counter();
ctr.increment();
ctr.increment();
```

When the first call to the method `increment` has finished, the abstract value of the ghost field `count` becomes 1. After the second call, its value becomes 2. This means that the surrogate object of the interface `Countable` for the object pointed to by the variable `ctr` changes its state. The consequence of this is that the runtime assertion checker cannot create a fresh surrogate object each time it makes a dynamic call to an assertion method (e.g., the postcondition method of the method `increment`) of a surrogate class to inherit specifications from an interface unless the surrogate class is stateless. A unique surrogate object must be attached to each instance, and it has to be used for dynamic assertion calls for the lifetime of the instance. For example, a surrogate object for the interface `Countable` is created for the object `ctr`, and it should be used for all dynamic assertion calls for `ctr`, e.g., checking postconditions of the method `increment` specified in the interface `Countable`.

The JML compiler's approach to stateful interfaces is:

1. To associate an instance of an implementing class, as a delegee object, to each surrogate when the surrogate object is first created for dynamic assertion calls (see Section 5.5 for surrogate classes), and

2. To maintain a per-object mapping from interfaces to surrogates for subsequent dynamic assertion calls.

The first is for a surrogate to make static delegation calls correctly back to the objects being assertion checked (see Section 5.5 for details). The second, of course, is for the benefit of objects themselves being assertion checked.

Figure 6.10 shows code for managing the per-object mapping from interfaces to their surrogate objects. The code is added to each class that is compiled with runtime assertion checking. The focal method is the static method `rac$receiver`, which is called by the dynamic call method such as `rac$check` to determine the receivers of dynamic assertion calls. Given the name of a target class or interface, `name`, and the object being checked, `forObj`[2], the method determines the receiver for a dynamic call. If a dynamic call is made to an interface, the method looks for a surrogate object for the interface in the surrogate map, `rac$surrogates`. If one is found, it is returned; otherwise, the method creates a new surrogate object for the interface with the object being checked, `forObj`, as the delegee. For this, the method uses Java's reflection facility. The method then adds the created surrogate into the surrogate map and returns the surrogate as the result.

6.5.1 Propagating Assertion Calls to Superinterfaces

An interesting situation arises when an interface inherits specifications from its superinterfaces. The interface has to call the assertion methods of its superinterfaces' surrogate classes. If the assertion methods are instance methods, e.g., for inheriting the preconditions of instance methods, the interface has to make dynamic calls to surrogate objects of its superinterfaces. Thus, the runtime assertion checker has to create a new surrogate object on the fly or retrieve an existing one. It may not create a new surrogate object for the superinterface due to the statefulness of surrogate objects. Then, where should it look for the surrogate object for the superinterface? The assertion checking originates from an instance of a class implementing the interface that tries to inherit specifications from its superinterfaces. That instance, which is the delegee of the surrogate of the interface, is where the surrogate object for the superinterface has to be looked for; the instance is supposed to store a unique surrogate object for each interface it implements either directly or indirectly. If a surrogate object for the superinterface is not found, then the runtime assertion checker has to create a new surrogate with the instance as the delegee, and register the new surrogate to the instance's store for later use. This happens when it is the first time that the instance inherits instance specifications from the superinterface. As an example, suppose that S is a class that directly implements an interface, I_1, which directly extends an interface, I_2, which directly extends an interface, I_3, and so on up to an interface I_n. Then, each instance of the class S, say, o_i, has a map $\{I_j \mapsto s_j\}$, where $j = 1, \ldots, n$ and s_j is a surrogate for the interface I_j. Each surrogate object, s_j, has the object, o_i, as its delegee object.

The above scheme is materialized by the method `getReceiver` defined in the class `JMLSurrogate` (see Figure 6.11). The class `JMLSurrogate` is the common superclass of all surrogate classes. The method `getReceiver` determines the receiver for a dynamic assertion call, given the target type, `clazz`, and the requesting object, `forObj`. As explained before, the method first looks up a surrogate object in the delegee's surrogate map. If no surrogate is found, the method dynamically creates a new one with this surrogate's delegee as the delegee by using Java's reflection facility. The method then adds the created surrogate to the delegee's surrogate map.

6.5.2 Lazy Initialization of Surrogate Maps

In Figure 6.10, the declaration of a surrogate map, `rac$surrogates`, has no initializer. Instead, it is initialized lazily by the access methods `rac$surrogate` and `rac$setSurrogate`. This lazy initialization is critical for the correct operation of the runtime assertion checker. The reason being that often a surrogate object needs to be created before the initialization of fields (e.g., `rac$surrogates`) are completed. In Java, for example, an explicit or implicit invocation of a superclass constructor

[2] If the argument `forObj` is null, it means that the dynamic call is made to a static method.

```
public java.util.Map rac$surrogates;

public Object rac$getSurrogate(String name) {
  if (rac$surrogates == null) {
    rac$surrogates = new java.util.HashMap();
    return null;
  }
  return rac$surrogates.get(name);
}

public void rac$setSurrogate(String name, JMLSurrogate obj) {
  if (rac$surrogates == null) {
    rac$surrogates = new java.util.HashMap();
  }
  rac$surrogates.put(name, obj);
}

public static Object rac$receiver(String name, Object forObj) {
  if (forObj == null || !name.endsWith("$Surrogate")) {
      return forObj;
  }
  //@ assume forObj instanceof JMLCheckable;
  JMLCheckable cobj = (JMLCheckable) forObj;
  try {
    Object surObj = cobj.rac$getSurrogate(name);
    if (surObj == null) {
      java.lang.Class[] types = new java.lang.Class[] { JMLCheckable.class };
      java.lang.Class clazz = java.lang.Class.forName(name);
      java.lang.reflect.Constructor constr = clazz.getConstructor(types);
      Object[] args = new Object[] { cobj };
      surObj = constr.newInstance(args);
      cobj.rac$setSurrogate(name, (JMLSurrogate)surObj);
     }
     return surObj;
  }
  catch (Exception e) {
    //@ unreachable;
  }
  return null;
}
```

Figure 6.10: Maintaining per-object surrogate maps for classes.

```
protected static Object getReceiver(Class clazz, Object forObj)
  throws ClassNotFoundException {
  if (forObj == null) { // dynamic call to static methods?
    return null;
  }
  Object delegee = null;
  if (forObj instanceof JMLSurrogate) {
    delegee = ((JMLSurrogate) forObj).delegee();
  } else {
    if (JMLSurrogate.class.isAssignableFrom(clazz)) {
      delegee = forObj;
    } else {
      return forObj;
    }
  }
  JMLCheckable cobj = (JMLCheckable) delegee;
  try {
    Object surObj = cobj.rac$getSurrogate(clazz.getName());
    if (surObj == null) {
      Class[] types = new Class[] { JMLCheckable.class };
      Constructor constr = clazz.getConstructor(types);
      Object[] args = new Object[] { cobj };
      surObj = constr.newInstance(args);
      cobj.rac$setSurrogate(clazz.getName(), (JMLSurrogate)surObj);
    }
    return surObj;
  }
  catch (Exception e) {
    //@ unreachable;
  }
  return null;
}
```

Figure 6.11: Determining surrogate objects for dynamic calls. Defined in the class `JMLSurrogate`, the common superclass of all surrogate classes, the method `getReceiver` determines the receiver object (a surrogate object) for a dynamic assertion call. The argument `clazz` is the target class for dynamic call and the argument `forObj` is the requesting object.

(`super()`) in a constructor precedes the execution of the instance initializers a class [55, Section 12.5].

Figure 6.12 shows an example that illustrates the need for such a lazy initialization. If one creates an instance of the class `S` by the statement `new S()`, the `super` statement of the constructor is executed before the surrogate map field, `rac$surrogates`, added by the JML compiler, of the class `S` is initialized. Now, the constructor of the class `T`, the direct superclass of `S`, has a `set` statement in the body. The `set` statement refers to the ghost field `val` defined in the interface `I`. To access the ghost field, the runtime assertion checker has to look up a surrogate object for the interface `I`. Without the lazy initialization, this lookup will encounter a runtime exception as the surrogate map is not initialized yet.

```
public class S extends T {
  public S(int x) {
    super(x);
    //@ assert val == x;
  }
}

class T implements I {
  public T(int x) {
    //@ set val = x;
    // ...
  }
}

interface I {
  //@ instance ghost int val;
}
```

Figure 6.12: Example illustrating the need for lazy initialization of surrogate maps.

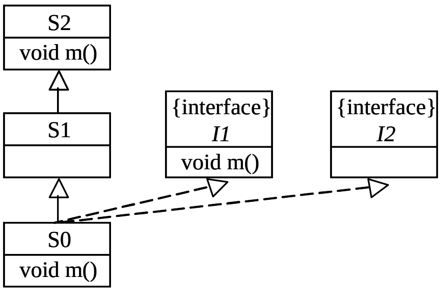

Figure 6.13: Example of specification inheritance.

6.6 An Example

Figure 6.13 shows a simple class hierarchy involving three classes and two interfaces. The class S0 overrides an instance method, m, inherited from the superclass S2, and it also implements both interface I1 and I2. The interface I1 declares the method m. It is assumed that each declaration and definition of m has a method specification, which is not shown in the diagram. Figure 6.14 shows a sequence of assertion method calls to be performed when the method m is called upon an instance of class S0, e.g., new S0().m(). The method m inherits such method specifications as preconditions, normal postconditions, and exceptional postconditions only from the class S2 and the interface I1. For instance invariants and constraints, however, it inherits from all superclasses and interfaces, in particular including the class S1 and the interface I2.

6.7 Discussion

The delegation approach to specification inheritance supports multiple inheritance and separate compilation. For separate compilation, the only requirement is to be able to determine at compile-time the set of immediate supertypes for specification inheritance. For a given method, the set of

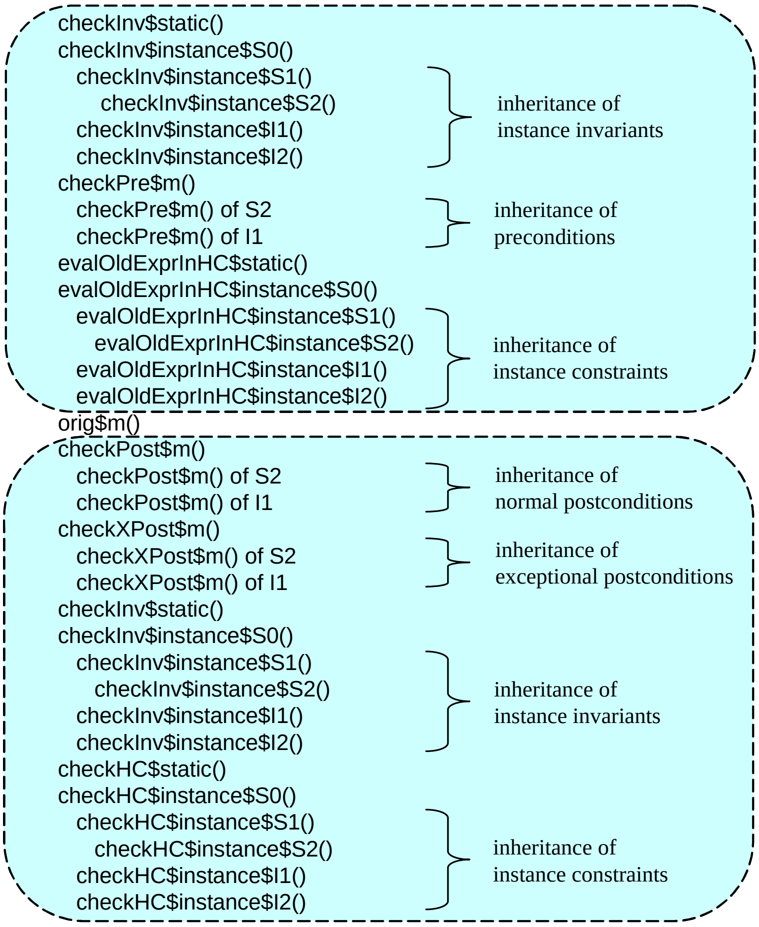

Figure 6.14: Assertion method calls for specification inheritance. This is the sequence of assertion method calls to be performed when the method `m` is called upon an instance of class `S0` shown in Figure 6.13.

immediate supertypes can be determined without source files, and in that sense the approach is modular; the decision can be based solely on the information about method signatures and type hierarchy, which is available from from Java bytecode files.

In the following, I discuss some aspects of specification inheritance, including an anomaly of specification inheritance, checking behavioral subtyping, and inheritance through refinement.

6.7.1 Anomaly of Specification Inheritance

In JML, a class inherits specifications from the interfaces that it implements; only specifications are inherited, as there is no code to inherit from interfaces. This can introduce an interesting situation, depicted in Figure 6.15. The essence of the problem is that the method `m` defined in the superclass `T` has to implement the specification of the interface `I` that the subclass `S` implements. For the method `m`, the subclass `S` inherits both specifications and code from the superclass `T`, but only specifications from the interface `I`. As the subclass `S` implements the interface `I`, the inherited code of the method `m` has to implement the declaration given in the interface `I`. In particular, it has to satisfy the method specification written in the interface `I`. However, assertion methods for `m` such

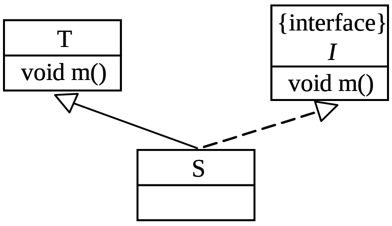

Figure 6.15: Inheriting only specifications. For the method `m`, the subclass `S` inherits both specifications and code from the superclass `T`, but only specifications from the interface `I`. The problem is that the inherited code of `T` has to satisfy the specification inherited from the interface `I`.

as pre- and postcondition methods are all defined in the superclass `T`, and their code do not make dynamic assertion calls to the interface `I` because the class `T` does not implement the interface `I`. Thus, method `m`'s specification inherited from the interface `I` is not checked even if the receiver is an instance of the subclass `S`, e.g., `new S().m()`.

One approach would be to add the following method in the subclass `S`. That is, such a method is overridden in the subclass so that it can be compiled by the JML compiler in the subclass with appropriate dynamic assertion calls for inheriting specifications from the interfaces that the subclass implements.

```
public void m() { super.m(); }
```

A similar sort of problems exists for type assertions such as invariants and constraints. A method, say `m`, inherited from the superclass `T` but not overridden in the subclass `S` still have to satisfy type assertions of the subclass `S` and the interface `I` that the subclass `S` implements. A complication in terms of runtime assertion checking is that the type assertions of `S` and `I` should be checked only if the object under checking is an instance of `S` (or its subclasses), e.g., `new S().m()`; they should not be checked if the object is an instance of `T`, e.g., `new T().m()`.

As before, one approach would be to add a simple delegation method in the subclass; however, this would result in adding a lot of such methods in the subclass, as one should be added for each inherited method. A better approach would be to provide down calls for type assertion methods by judiciously naming them. The idea is for the supertype's wrapper methods to call the subtype's type assertion methods through down calls. For example, the JML compiler may add assertion methods for down calls, e.g., `checkInv$instance`, in addition to the type-specific assertion method, e.g., `checkInv$instance$S`, as follows.

```
public void checkInv$instance() { checkInv$instance$S(); }
public void checkInv$instance$S() { /* ... checkInv$instance$T() ... */ }
```

As the first (down call) method has the same name for all types, the subtype's overrides the supertype's, thus allowing down calls from the supertype to the subtype. The second method, as before, is for a subtype to inherit specifications from its supertypes. With this setting, method calls like `new S().m()` now checks invariants correctly, including those specified in the subclasses. The wrapper method for `m` added to the superclass `T` calls the (down call) invariant method declared in `T`. However, what is invoked is the subtype `S`'s (down call) invariant method, because it is overridden in `S`. Thus, all relevant invariants are checked. The same technique works for constraint methods.

The anomaly of specification inheritance affects modular verification of object-oriented programs [84] [91] [92] [95] [114] [113] [151], as the supertype's methods, if not overridden, have to be verified again with respect to the subtype's specifications. This may mean that the supertype's implementation should be available for such re-verification. However, the implications of the anomaly on verification and reasoning need to be thoroughly investigated in the future.

6.7.2 Checking Behavioral Subtyping

JML supports the concept of behavioral subtyping [1] [84] [91] [92] [101], by imposing the specifications of supertypes on their subtypes [40]. In particular, the inherited preconditions are disjoined to the explicitly specified ones, and the inherited postconditions are conjoined to the explicitly specified ones. The postconditions need to hold in the post-state only when the corresponding preconditions holds in the pre-state; i.e., the effective postcondition has the form, `\old(`P_i`)` $\Rightarrow Q_i$, where each P_i is a precondition and Q_i is the corresponding postcondition (see Section 6.2.3). A pleasant consequence of this decision is that checking violations of behavior subtyping [49] turns into checking violations of inherited specifications, as behavioral subtyping is guaranteed to hold by the semantics of JML specification inheritance. That is, the runtime assertion checker checks behavioral subtyping by being faithful to the JML semantics of specification inheritance imposing the specifications of supertypes on the subtypes. As explained in this chapter, JML also supports weak behavioral subtyping where a subtype's additional methods are relieved from satisfying its supertype's constraints [38] [39] [40]. Weak behavioral subtyping is also checked by the runtime assertion checker.

6.7.3 Refinement

In JML, specifications are also inherited through refinement chains [89]. For example, the following `refine` declaration states that the class `T` in the current compilation unit (e.g., file `T.java`) "refines" the same class in the file `T.jml`. The class `T` in the current compilation unit is called a *refining type* and the one in `T.jml` is called a *refined type*. These terms are relative, as a reined type may refine another type, thus becoming a refining type itself.

```
//@ refine "T.jml";
public class T { /* ... */ }
```

The meaning of refinement is that a refining type inherits all specifications and code of the refined type, including even private and package-visible ones. The JML's notion of refinement allows one to write both code and specifications incrementally, e.g., partial code and specifications in the refined type and detailed, complete code and specifications in the refining type. It also provides a good structuring tool for specifications; specifications can reside in separate files from code files, and can be attached to existing source code and even to bytecode.

However, refinement poses a challenge to the runtime assertion checker. Each type in a refinement chain is not a complete program type that have a separate identity in the final implementation code. Rather, it consists of pieces of code and specifications that need be merged to other types (i.e., refining types) to become a complete type. The refining and refined types have the same name. A consequence of this is that there is no Java program class or interface that can host assertion methods of the refined and refining types separately.

There are at least two approaches to solving this problem: a textual copy approach and a surrogate approach. In the *textual copy approach*, all the specifications of the refined type are copied down to the refining type, and compiled as if they were written in the refined type. That is, all code and specifications of types in a refinement chain are combined into a one type before being compiled. However, the approach is unpalatable in that it does not support modular compilation and may require complex renaming due to potential name clashes between refined and refining types. In addition, it is not clear whether the refined and refining types can be successfully combined, at the source code and specification level, into one type in all possible circumstances; different compilation units may have different contexts such as import declarations.

In the *surrogate approach*, a separate surrogate class is generated for each refined type, just like a surrogate class for an interface. The surrogate class is responsible for checking assertions specified in the refined type. A refining type uses dynamic assertion calls, similar to those for inheriting specifications from interfaces, to inherit specifications from the refined type. In this case, however, specifications of all privacy levels are inherited by the refining type; thus, implementing the surrogate approach may not be straightforward. The surrogate approach can support modular compilation provided that some pertinent specification information is encoded into bytecode, and it does not require complex renaming. Furthermore, it has a merit of treating specification refinement in the same way as in inheriting from interface specification, thereby providing a seamless, universal framework for specification inheritance.

The JML compiler does not support refinement yet, and it is not decided yet which of the above two approaches would be supported by the JML compiler.

Chapter 7

Abstract Specifications

The use of abstract, specification-purpose fields and methods in JML can be seen as an evolutionary advance over the design-by-contract approaches. The main advantage is that one knows exactly what parts of the code are only for specification purposes. The role of each declaration becomes clearer both to the reader and the tools. In this chapter, I explain how the JML compiler supports specification-purpose declarations in JML such as model fields, ghost fields, and model methods.

The JML compiler's approach is to use access methods in combination with dynamic calls. The key idea is to generate an access method for a specification-purpose declaration and to translate each reference to the declared member into a dynamic call to the corresponding access method. For example, an access method for a model field is generated from the abstraction function for the field, specified by the `represents` clause; the access method calculates (or retrieves) an abstract value from the program state. For a ghost field, a pair of getter and setter access methods is generated so that the field can be directly manipulated by using specification statements. For an executable model method, the access method has the body of the model method; for a model constructor, the access method becomes a `static` factory method. The JML compiler also generates a default form of access methods that throw angelic undefinedness, e.g., for model methods with no abstraction functions. If dynamic calls to access methods fail, they become angelic undefinedness. The JML compiler's approach supports separate compilation and is faithful to the semantics of JML.

7.1 Introduction

7.1.1 Abstraction in JML

JML provides several specification-purpose declarations such as model fields, ghost fields, model methods, and model classes and interfaces [89]. They have the same syntax as in Java except for the additional JML modifiers `model` and `ghost`. The specification-only members can be used only in specifications; they cannot be used in program code. They do not have to appear in an implementation, and thus are important tools for improving the abstraction level of specifications.

The values of specification-purpose fields are abstract in the sense that one is not concerned with their time or space efficiency, but more with clarity [62] [63] [100]. A model field's value is given as a mapping from the program state, whereas a ghost field's value is directly manipulated with specification statements. A `represents` clause defines this mapping for a model field by stating how its value is related to the program state [50] [89] [95] [96]. The `set` specification statement, similar to Java's assignment statement, is used to set the value of a ghost field.

A model method is a method declared solely for the purpose of writing specifications, and is not required to have a body. A model method is often used to abbreviate some specification expression

```
public class Timer {
  private long sec = 0;
  //@ public model long min;
  //@ private represents min <- sec / 60;

  //@ ensures \result == min;
  public long getMin() { return sec / 60; }
  // ...
}
```

Figure 7.1: Example of Model fields with abstraction functions.

or to define an abstraction function for a model field. Similarly, a model class or interface is declared solely for the purpose of writing specifications.

7.1.2 General Approach

The JML compiler supports both specification-purpose fields and methods. However, it does not support model classes and interfaces yet. The main requirement is to support separate compilation, as a specification-purpose field or method declared in one source file may be used in another source file. The essence of the JML compiler's approach is to use dynamic calls and access methods. Each reference to a specification-only member is replaced with a dynamic call to the access method of the member. The access method knows how to interpret an access to the member, and is automatically generated by the JML compiler.

For example, the JML compiler views a model field as a function from the program state to the abstract values. Consider the model field `min` shown in Figure 7.1. The model field `min` denotes the current time of a timer in minutes. A model field may be accompanied by a `represents` clause that states how the field is related to program fields. The `represents` clause for the model field `min` states that the value of `min` is that of the expression `sec / 60`. Thus, it specifies an abstraction function for the field `min`. The JML compiler materializes this abstraction function into the access method for the field `min`, as follows. (The code is simplified, as the contextual interpretation code is suppressed.)

```
public long model$min() { return sec / 60; }
```

With this in place, each reference to the model field `min` is translated into a call to the access method `model$min`. Thus, the runtime assertion checker can evaluate assertions written in terms of `min`, e.g., the postcondition of the method `getMin`. As both model fields and `represents` clauses can be inherited by subtypes, the access methods are often called dynamically by using a mechanism similar to that described in Chapter 6. However, as a `represents` clause may be specified in subtypes, the approach is more complicated (see Section 7.2.3).

In the following sections, I explain how the JML compiler translates model fields, ghost fields, and model methods. Section 7.2 explains translation of model fields declared in classes. Section 7.2 discusses model fields declared in interfaces; inheritance is examined in depth. Section 7.4 and Section 7.5 explains translation of ghost fields and model methods respectively. Section 7.6 discusses some limitations and widening of Java visibility (e.g., `spec_public` and `spec_protected`).

7.2 Model Fields

The approach to implementing model fields is to generate a unique model field access method for each model field and to replace each occurrence of a model field in assertions with the corresponding access method. The model field access method retrieves, for the model field, an abstract value from the current program state. Thus, the translation of model fields consists of: (1) generating model field access methods, and (2) substituting the model field access method for each occurrence of a model field in assertions.

7.2.1 Model Field Access Methods

An access method for a model field is generated either from the field's declaration or from the field's abstraction function. If no abstraction function is specified for the field in the type where it is declared, then its access method is generated from its declaration. For such a model field, declared as "`model` T m`;`" in the type S, the JML compiler generates the following access method[1].

`public` T `model$`m`$`S`() { throw new JMLAngelicException(); }`

This form of access methods is called a *default model field access method.* As it throws an angelic exception, any reference to the field are interpreted angelically by its parent expressions (see Section 3.2.3 for the contextual interpretation). That is, the model field becomes non-executable. The access method is added to the host class of the model field, the class S or its surrogate class (if S is an interface).

There are two ways of relating program states to abstract values of model fields: *abstraction functions* [63] [100, pp. 70–71] and *abstraction relations* [6] [45] [92] [138] [140]. In JML, both are specified by using `represents` clauses. The JML compiler translates abstraction relations into the default form of model field access methods. This means that all references to model fields with abstraction relations (e.g., `represents` m `\such_that` P`;`) are treated as angelic undefinedness, i.e., they are non-executable.

The JML compiler translates abstraction functions into model field access methods. For example, consider a `represents` clause of the form, "`represents` m `<-` E`;`", where the model field m is declared in the type S and has the type T. Then, the JML compiler generates the following model field access method.

`public` T `model$`m`$`S`() {` T `rac$`v`;` $[\![E,$ `rac$`$v]\!]$ `return rac$`v`; }`

The access method computes an abstract value for the model field m by evaluating the abstraction function specified by the `represents` clause. The notation $[\![E, \mathrm{rac\$}v]\!]$ denotes the result of translating the expression E into Java program code that evaluates E and stores the result into `rac$`v; it is a short-hand notation for $\mathcal{C}[\![E,$ `rac$`v, `false`$]\!]$ (see Chapter 3 for the translation function $\mathcal{C}[\![\cdot]\!]$. The access method is added to the host class of the `represents` clause. The host class of the `represents` clause clause may not be the same as the type where the model field itself is declared. Therefore, it is possible for a model field to have more than one access method, and the one in the subtype overrides the one in the supertypes.

7.2.2 Assertions with Model Fields

The next step of the translation is to substitute an appropriate access method call for each occurrence of a model field in assertions. Suppose a JML expression of the form, E`.`m, where E is an expression of type, say, T and m is a model field declared in a type S[2]. Then, the expression E`.`m is translated into the following piece of code.

T `rac$`v`;` $[\![E,$ `rac$`$v]\!]$ `rac$`v`.model$`m`$`S`();`

First, the expression E is evaluated and the result is saved into a fresh local variable, `rac$`v. Then, instead of referencing the model field m of `rac$`v, m's access method `model$`m`$`S is called

[1] If the model field is static, its access method becomes static.

[2] T should be either S or one of its subtypes.

```
$\mathcal{C}$: Expression $\times$ Identifier $\times$ boolean $\rightarrow$ Program$\mathcal{C}[\![E.I, r, p]\!] \overset{\text{def}}{=} \underline{?\ \text{if } I\text{'s type is boolean}}$
  try {
    $\mathcal{C}_R[\![E.I, r, p]\!]$
  } catch (JMLAngelicException $e$) {
    $r$ = !$p$;
  } catch (Exception $e$) {
    $r$ = $p$;
  }

$\mathcal{C}[\![E.I, r, p]\!] \overset{\text{def}}{=} \underline{?\ \text{otherwise}}$
  $\mathcal{C}_R[\![E.I, r, p]\!]$

$\mathcal{C}_R$: Expression $\times$ Identifier $\times$ boolean $\rightarrow$ Program
$\mathcal{C}_R[\![E.I, r, p]\!] \overset{\text{def}}{=} \underline{?\ I \text{ is a model field declared in } S}$
  $T$ $v$;
  $\mathcal{C}[\![E, v, p]\!]$
  $r$ = $v$.model$$I$$S$();

$\mathcal{C}_R[\![E.I, r, p]\!] \overset{\text{def}}{=} \underline{?\ \text{otherwise}}$
  $T$ $v$;
  $\mathcal{C}[\![E, v, p]\!]$
  $r$ = $v$.$I$;
```

Figure 7.2: Extended translation rules for field reference expressions.

upon rac$$v$. Though not shown in detail in the above code, the access method may be invoked dynamically using Java's reflection facility [146]. If the model field is declared in the same source file, the access method is invoked statically; otherwise, it is invoked dynamically to support separate compilation. Figure 7.2 shows the translation rules for field reference expressions, originally defined in Section 3.2.4 (see Figure 3.3), extended for model fields.

There is an interesting aspect in translating model field references into access method calls. Because it is possible for a subtype's model field to hide an inherited model field, the runtime assertion checker has to be precise about which declaration each field reference refers to. Suppose that both a class, S, and its superclass, T, declare a public model field, m. Then, the one in the subclass hides the one in the superclass [55, Section 8.3]. In the subclass S, a model field reference of the form, m, this.m, or $E.m$, where E's static type is S, is translated into a call to the access method added to the subclass, i.e., model$$m$$S$. On the other hand, a reference of the form, super.m and E.m, where E's static type is T, is translated into a call to the access method added to the superclass T, i.e., model$$m$$T$. However, note that the subclass S may also provide an access method for the hidden model field declaration of the superclass T (i.e., model$$m$$T$) by specifying an abstraction function for it (e.g., represents super.m <- E;). In that case, all calls to the access method model$$m$$T$ invoke the one added to the subclass S because the subclass' overrides the one inherited from the superclass T (refer to Section 7.2.3 for the inheritance of model fields).

In sum, the translation of model field references reflects Java's semantics of statically resolving field names, and yet supports JML's semantics of abstraction functions that may be specified separately from the model field declarations.

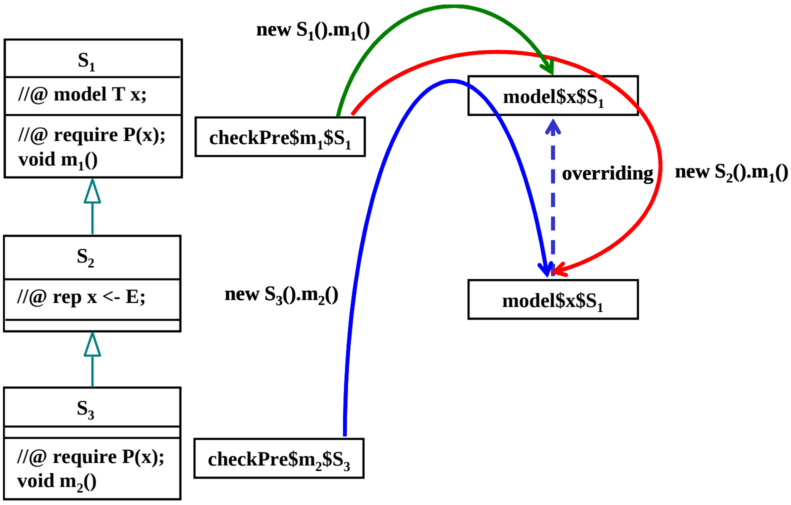

Figure 7.3: Inheritance of model fields.

7.2.3 Inheritance

Model fields obey the same rules as Java's program fields do [89]. In particular, subclasses inherit model instance fields from superclasses, and may hide them. In addition to model fields, subclasses also inherit instance `represents` clauses from superclasses. The translation rules presented in the previous sections support inheritance of both model fields and `represents` clauses. The correctness of the translation rules regarding inheritance relies on dynamic calls and the semantics of Java's dynamic invocation. The main reason for using dynamic calls, with the help of Java's reflection facility [146], is to support separate compilation.

Figure 7.3 shows an example inheritance of model fields, involving three classes S_1, S_2, and S_3. The class S_1 is a superclass of S_2, and S_2 is a superclass of S_3. The class S_1 declares a model field, $v$, whose abstraction function is specified in the subclass S_2. The model field $v$ is used in assertions of the class S_1 and S_3. As the class S_1 does not specify an abstraction function for the model field $v$, a default access method, `model$`v`$`S_1, is added to S_1. The `represents` clause in the subclass S_2 also triggers the generation of an access method, `model$`v`$`S_1, for $v$ in S_2. As the result, there are two access methods for the model field $v$, a default one in the class S_1 and an abstraction function in the subclass S_2. As both have the same name and signature, the one in the subclass overrides the one in the superclass.

Now, let us consider occurrences of the model field $v$ in various assertions. First, consider the use of $v$ in the subclass S_3, e.g., the precondition of the method m_2. Remember that the class S_3 inherits the model field $v$ from the class S_1 and $v$'s abstraction function from the class S_2. Therefore, an occurrence of $v$ in the class S_3 must be interpreted using the abstraction function inherited from the class S_2. Does the runtime assertion checker live up to this expectation? According to the translation rules, the precondition method of m_2, `checkPre$`m_2`$`S_2, dynamically calls the model field $v$'s access method, `model$`v`$`S_1. For this, it first looks up an access method of $v$, using Java's reflection facility [146], in the class S_1, where the field $v$ is declared. As the class S_1 has a default access method for $v$, the precondition method certainly finds one provided that S_1 is compiled with the JML compiler. Then, the precondition method calls the access method using Java's reflection facility. The semantics of Java's dynamic invocation is that if the called method is overridden by a subclass, then invoked is the overriding method of the subclass [146]. That is, reflective method calls also have the dynamic dispatch semantics [55, Section 8.4.6 and 15.12]. Thus, if the receiver is an instance of the class S_3, then invoked is the access method added to the class S_2. Therefore, the

```
//@ model import org.jmlspecs.models.JMLObjectSequence;
public interface StackType {
  /*@ model instance non_null JMLObjectSequence elems
    @  initially elems.isEmpty(); @*/

  /*@ assignable elems;
    @ ensures elems.equals(\old(elems.insertBack(e))); @*/
  void push(/*@ non_null @*/ Object e);

  // ...
}
```

Figure 7.4: Example of interface model fields.

translation is faithful to the semantics of JML regarding inheritance of model fields and represents clauses.

A more interesting situation arises for an occurrence of the model field v in the method m_1's precondition in the class S_1 (see Figure 7.3). Here, the abstraction function specified in the class S_2 may not be used for the interpretation of the model field v. If the runtime (or dynamic) type of the object being checked is S_1, then the abstraction function specified in S_2 cannot be used for the interpretation of v. Instead, such an occurrence of v must become non-executable. On the other hand, if the runtime type is S_2 or one of its subclasses, the abstraction function specified in the class S_2 must be used for the interpretation. How does the runtime assertion checker achieve this? As before, the precondition method of m_1, checkPre\$$m_1$\$$S_1$, dynamically calls the access method of v, model\$$v$\$$S_1$. If the receiver object's runtime type is S_1, the call goes to the default access method added to the class S_1. If the runtime type is S_2 or one of its subclasses, the call goes to the access method added to the class S_2; this again is due to the dynamic dispatch semantics of Java's reflective method calls. This is an example of the so-called a *down call*, a superclass' call to a subclass' overriding method. As illustrated by the example, it is crucial for access methods for the same model field to have the same name and signature, and for the runtime assertion checker to look up access methods in the class where the corresponding model fields are declared.

7.3 Interface Model Fields

An *interface model field* is a model field declared in an interface. An interface model field is an elegant way of specifying the behavior of an interface. The behaviors of interface methods can be described in a model-oriented style by referring to the abstract model provided by interface model fields. Figure 7.4 shows an example of interface model fields. A stack is modeled as a sequence of objects by the model instance field elems, and the behavior of the method push is described in terms of that model field. In the declaration of elems, the modifier instance indicates that the declared model field is non-static. As in Java, an interface model field by default becomes a static field. The initially clause at the end of the model field declaration permits the field to have an abstract initialization [112] [118]. The abstract initialization allows one to reason about the field by using data type induction [63].

An interface model field can also be accompanied with a represents clause. However, an interface represents clause is often written in terms of other model fields, as no program fields except for final static fields can be declared in an interface. This means that an interface assertion written with model fields becomes executable only if all the model fields are eventually mapped to the program states by the implementing class. It is often the case that an interface does not provide abstraction functions for its model fields. Instead, an implementing class of the interface provides abstraction functions for the interface model fields. The executability of interface model fields can also be achieved by a layer of abstraction functions; e.g., an interface model field is represented by another interface model field whose abstraction function is defined by the implementing class.

7.3.1 Inheritance

An interface's model instance field is inherited by an implementing class and an extending interface. An instance `represents` clause appearing in an interface is also inherited by an implementing class and an extending interface. What is the meaning of inheriting a `represents` clause? If an implementing class or an extending interface refers to an inherited interface model field, the model field must be interpreted by using the abstraction function specified by the inherited `represents` clause. Figure 7.5 shows an example specification where interface model fields and `represents` clauses are inherited through interfaces and subclassing. The interface `I1` declares a model instance field `val1`, which is inherited by all implementing classes and subinterfaces, in particular, the subinterface `I2`. The subinterface `I2` declares an additional model instance field of its own, `val2`. In addition, it also specifies an abstraction function for the inherited model field `val1`. Both model fields and the abstraction function are inherited by the interface `I2`'s implementing classes such as the class `S1` and its subclass `S2`. The subclass `S2` provides an abstraction function for the inherited interface model field `val2`. From the perspective of the class `S2`, both model fields are executable; that is, their abstract values are well defined in terms of the program state. The value of the model field `val2` is the same as that of the program field `cval`, as specified by the `represents` clause of the class `S2`. The value of the model field `val1` is the same as that of `val2`, as specified by the `represents` clause of the interface `I2`, thus that of the program field `cval` of the class `S2`. Therefore, the precondition of the method `m2` of the class `S2`, which is written in terms of the model field `val1` can be evaluated by the runtime assertion checker. However, from the perspective of the class `S1`, none of the model fields is executable, as the abstraction function for the model field `val2` is not provided. An interesting use of the model field `val1` is the one that appears in the precondition of the method `m1` specified in the interface `I1`. Should such an appearance be interpreted by using the abstraction function specified by the interface `I2`? It depends on the object being assertion checked. If the object is an instance of a class that implements the interface `I2`, that abstraction function must be used, as the `represents` clause of `I2` is inherited by that class. Otherwise, that abstraction function must not be used, as the `represents` clause of `I2` is not inherited by that class. For the same reason, the abstraction function specified in the class `S2` for the model field `val2` should be used only if the object being checked is an instance of the class `S2`. As a corollary, the runtime assertion checker can retrieve the abstract value of the model field `val1` appearing in the precondition of the method `m1` only if the object being checked is an instance of the class `S2`.

How does the runtime assertion checker supports the inheritance of interface model fields and `represents` clauses? This is a challenging task, and its solution is one of the many contributions of this dissertation. One complication is that an applicable abstraction function (i.e., a `represents` clause) must be determined at compile-time for each occurrence of an interface model field in interface assertions. Under separate compilation, however, the applicable abstraction function cannot be determined statically at compile-time because it may be specified by a subinterface or an implementing class. A similar problem exists for a model field declared in a class, for which the runtime assertion checker relies on Java's inheritance mechanism to implement the so-called down call (see Section 7.2.3). For an interface model field, however, the runtime assertion checker cannot completely depend on Java's inheritance mechanism because Java does not allow multiple inheritance for classes; multiple inheritance of interfaces results in multiple inheritance of classes, as each interface generates a corresponding surrogate class. Another complication is that there are now several kinds of up-calls and down-calls: (1) between classes, (2) between classes and interfaces, and (3) between interfaces.

To recap, the main problem is to accomplish the effect of down calls in the presence of multiple

```
public interface I1 {
  //@ public model instance int val1;

  //@ requires val1 > 0;
  void m1();
}

public interface I2 extends I1 {
  //@ public model instance int val2;
  //@ public represents val1 <- val2;
}

public class S1 implements I2 {
  public void m1() {}
}

public class S2 extends S1 {
  private int cval = 0;
  //@ private represents val2 <- cval;

  //@ requires val1 > 0;
  public void m2() {}
}
```

Figure 7.5: Inheritance of interface model fields.

inheritance of interfaces. A key challenge is for a superinterface to prepare to use abstraction functions that may be provided by its subinterfaces. The JML compiler's approach is to introduce two additional model field access methods called a *dispatch method* and a *delegation method* respectively (see Figure 7.6). A dispatch method (e.g., `model$`v`$`I_1 of S) is added to an implementing class for each inherited interface model field. The dispatch method is responsible for invoking an appropriate model field access method (e.g., `model$`v`$`I_1 of I_2) that is generated from an interface's `represents` clause and added to the surrogate class of that interface. Because the dispatch method is added to an implementing class, it is possible to statically determine whether an applicable interface `represents` clause exists. If there is no such an interface `represents` clause, the dispatch method throws an angelic undefinedness exception (see Section 3.2 for undefinedness). For each interface model field, a delegation method (e.g., `model$`v`$`I_1 of I_1) is added to the surrogate class of the interface. If the interface specifies a (functional) `represents` clause for the model field, no delegation method is generated; instead, an abstraction function is generated from the `represents` clause, e.g., `model$`v`$`I_1 of I_2. Otherwise, a delegation method is generated, and it delegates incoming calls to the corresponding dispatch method. Each occurrence of the interface's model fields in interface assertions is now translated into a call to the corresponding delegation method. Remember that the delegation method calls the dispatch method which calls an appropriate abstraction function provided by some interface if such an abstraction function exists. Thus, this translation scheme has the effect of super calls if an abstraction function is provided by a superinterface, and of down calls if an abstraction function is provided by a subinterface. Figure 7.6 shows an example of down calls. A reference to the model field v in the interface I_1 invokes the abstraction function specified by the subinterface I_2 through delegation and dispatch calls.

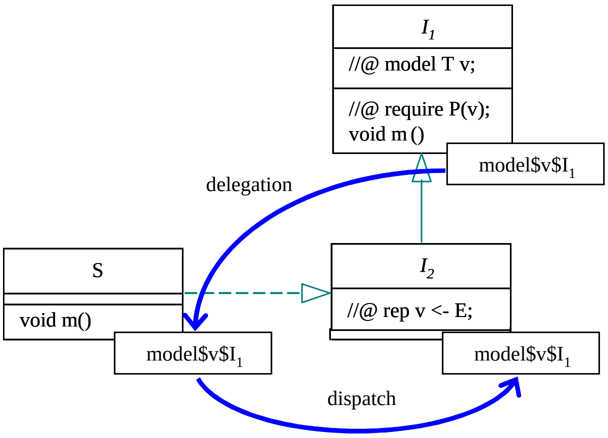

Figure 7.6: Achieving down calls to inherit interface model fields.

7.3.2 Details of the Approach

As usual, references to interface model fields are translated into dynamic calls to the appropriate access methods. However, the access methods to be invoked are statically determined at compile-time. The receivers of dynamic calls now may be surrogate objects of the interfaces where the model fields are declared or where their `represents` clauses appear. The approach can be summarized as follows.

1. A model field declaration in an interface generates a (model field) *delegation method* for the declared model field. The method delegates all incoming calls to the current object being checked, i.e., the delegee object. Each surrogate object has a reference to its delegee, which knows how to interpret the model field (see step 4).

2. A `represents` clause in an interface generates a (model field) access method for the represented model field. As in classes, such an access method evaluates the `represents` clause's expression and returns the result (see Section 7.2). The access method becomes an abstraction function for the model field.

3. An occurrence of a model field in interface assertions (including expressions of `represents` clauses) is translated into a dynamic call to the corresponding delegation method (see step 1). The target delegation method is statically determined at compile-time.

4. An interface's top-level implementation class generates a (model field) *dispatch method* for each interface model field except for those inherited through the superclass chain[3]. The dispatch method either (1) takes the default form throwing angelic undefinedness if the model field has no `represents` clause specified in the interface or (2) forwards incoming calls to the appropriate access methods generated from the interface's `represents` clauses (see step 2). The dispatch method may be overridden by a subclass, i.e., if the subclass specifies a `represents` clause for the model field. If the class itself specifies a `represents` clause for the model field, then no dispatch method is generated; instead, an abstraction function is generated from the `represents` clause.

Delegation Methods If an interface I declares a model field, say m, of type T, then an access method for the model field is added to the surrogate class of the interface I. If the interface also

[3] The superclass will generate dispatch methods for such model fields.

specifies an abstraction function for the model field, then a model field access method is generated from the abstraction function. Otherwise, the access method is a delegation method of the following form, where the notation $\mathcal{D}[\![E\texttt{.}m\texttt{()},r]\!]$ means that m is invoked dynamically on E using Java's reflection facility with the result being stored into r^4, and the method `getSelf()` returns the delegee of the surrogate object, i.e., the object being checked (see Section 5.5 for surrogate classes).

$\texttt{public } T \texttt{ model\$}m\texttt{\$}I\texttt{() \{ } T\ r\texttt{; } \mathcal{D}[\![\texttt{getSelf().model\$}m\texttt{\$}I\texttt{()},r]\!] \texttt{ return } r\texttt{; \}}$

Dispatch Methods Let S be a class with a superclass T. Let $M(S)$ and $M(T)$ respectively be the set of interface model fields that the class S and T inherit from its supertypes (superclasses and superinterfaces). Then, for each model field m in the set $M(S) - M(T)$, a dispatch method of one of the following forms is added to the class S. The model field m is assumed to be declared in the interface I and to be of type T.

$\texttt{public } T \texttt{ model\$}m\texttt{\$}I\texttt{() \{ throw new JMLAngelicException(); \}}$

$\texttt{public } T \texttt{ model\$}m\texttt{\$}I\texttt{() \{ } T\ r\texttt{; } \mathcal{D}[\![\mathcal{S}_{S,I_2}\texttt{(this).model\$}m\texttt{\$}I\texttt{()},r]\!] \texttt{ return } r\texttt{; \}}$

The first form is used if there is no abstraction function available for the model field m; i.e., there is no intermediate interface, through superinterface chains, between the class S and the interface I that provides a `represents` clause for the field m. As there is no abstraction function specified for the model field, the dispatch method throws an angelic undefinedness exception to treat any references to it as non-executable specification constructs[5]. If there is an abstraction function specified for the model field, say, by the interface I_2, between the class S and the interface I through superinterface chains, then the second form of dispatch method is used. The dispatch method calls the access method provided by the interface I_2 for the model field. In the notation $\mathcal{S}_{S,I_2}(\texttt{this})$, $\mathcal{S}_{S,I_2}$ is a function that maps each instance of the class S to a unique surrogate object of the interface I_2; it is an abstraction of the surrogate map described in Section 6.5 Thus, the notation $\mathcal{S}_{S,I_2}(\texttt{this})$ denotes the surrogate of the interface I_2 for the object `this`. As a surrogate may have its own state, each instance of a class is associated with a unique surrogate object for each interface that the class implements (see Section 6.5). In sum, a dispatch method delegates incoming calls to an appropriate access method generated from an interface's `represents` clauses, if such an access method exists; otherwise, it throws an angelic undefinedness exception, thus treating any reference to such a model field as a non-executable specification construct. The dispatch method follows the same naming convention as other access methods; thus it is overridden by a subclass if the subclass specifies a `represents` clause for the interface model field. That is, Java's inheritance mechanism is still used for the inheritance of model fields through subclassing.

References to Interface Model Fields Each reference to an interface model field is translated in the same way as in the translation of a model field declared by a class. The only difference is that now the receiver of an access method call must be a surrogate object, as the accessor methods are added to the surrogate class of the interface. As an example, suppose a model field access expression, $E.m$, where m is an model instance field declared in an interface, say I, and E is an expression with runtime type, say S, that implements the interface I. Then, the expression is translated into $\mathcal{S}_{S,I}(E)\texttt{.model\$}m\texttt{\$}I\texttt{()}$, where $\mathcal{S}_{S,I}$ is a function that maps each instance of the class S into a unique surrogate object of the interface I.

[4] The translation function, D : Expression × Identifier → Program, maps method call expressions into Java program code that makes dynamic method calls using Java's reflection facility and stores the result into the given variable.

[5] It is still possible for the class S or its subclasses provide an abstraction function for the model field.

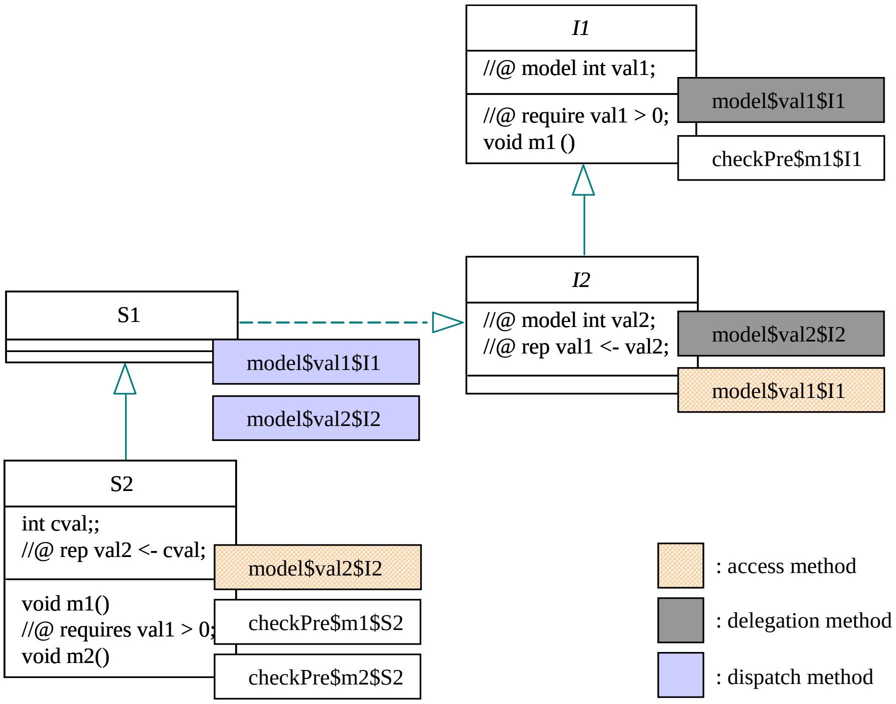

Figure 7.7: Access methods for interface model fields.

7.3.3 An Example

Figure 7.7 shows the classes and interfaces of Figure 7.5 with the added access methods for the interface model fields `val1` and `val2`. As the interface `I1` declares the model field `val1` without specifying its abstraction function, a delegation method for `val1` is added to the surrogate class of `I1`. The interface `I2` provides an abstraction function for the inherited model field `val1`; thus, a normal access method for the model field `val1` is added to the surrogate class of the interface `I1`. The interface `I2` declares an additional model model field, `val2`, without specifying an abstraction function. Thus, a delegation method for `val2` is added to the surrogate class of `I2`. The class `S1` is interesting. It implements the interface `I2`, thereby inheriting both the model fields `val1` and `val2`. As none of the model fields is also inherited through the superclass chain, dispatch methods for them are added to the class `S1`. The dispatch method for `val1` calls the abstraction function added to the surrogate class of `I2`, and the dispatch model for `val2` throws an angelic undefinedness exception. Finally, as the class `S2` specifies a `represents` clause for the inherited model field `val2`, an access method of `val2` is added to the class; note that the added access method overrides the dispatch method inherited from its superclass `S1`.

It would be instructive to examine the evaluation of a specific assertion to see the mechanics of model field access methods. Consider the precondition of the method `m1` specified in the interface `I1`, which is inherited by the method `m1` of the class `S2`. If one sends a message `m1` to an instance of the class `S2`, e.g, `new S2().m1()`, the runtime assertion checker performs a sequence of method calls to check the method `m1`'s specification including the preconditions (see Figure 7.8 and Figure 7.9). First, called is the precondition method of `m1` added to the class `S2`, i.e., `checkPre$m1$S2`. As the method `m1` inherits method specifications from the interface `I1`, the precondition method `checkPre$m1$S2` calls the corresponding precondition method of `I1`, `checkPre$m1$I1`, added to the surrogate class of the interface `I1`. To retrieve the value of the model field `val1` appearing in the precondition, the precondition method `checkPre$m1$I1` calls the access method (delegation method) of the model

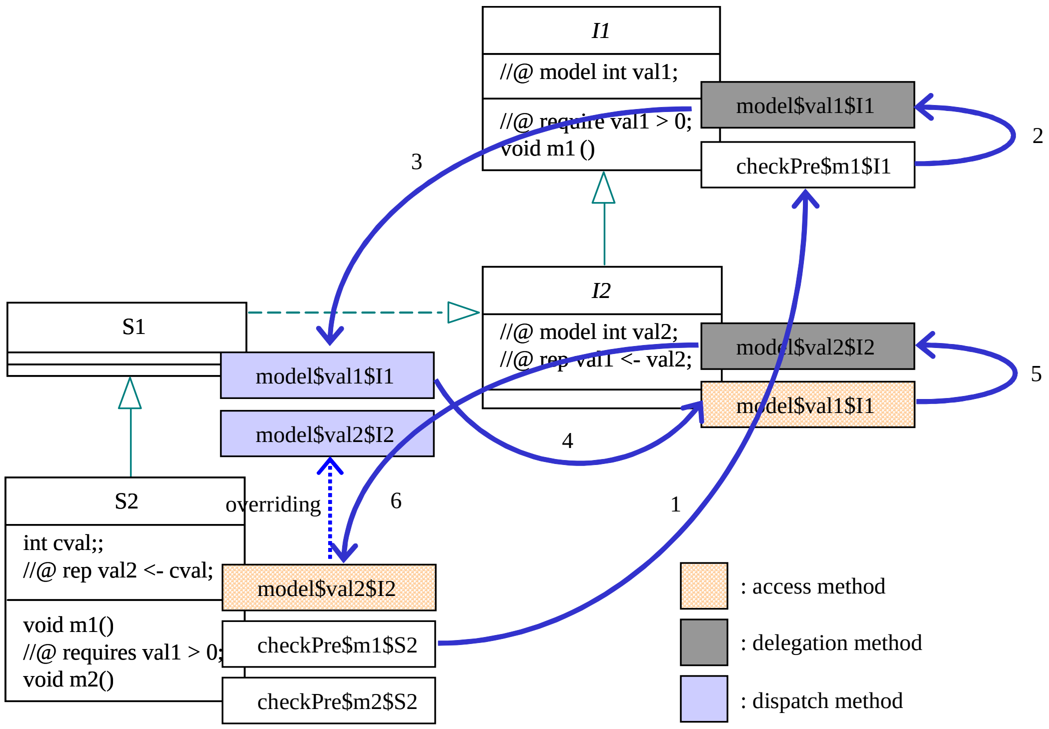

Figure 7.8: Sequence of access method calls to check the method `m1`'s precondition.

field `val1`, which delegates the call to the dispatch method of the class `S2`. The dispatch method calls the access method added to the surrogate class of the interface `I2`, as the interface `I2` specifies an abstraction function for the model field `val1`. Note that the abstraction function of `val1` is defined in terms of another model field, `val2`, declared in the interface `I2`. Thus, the evaluation of the abstraction function leads to a call to the access method for the model field `val2`, `model$val$I2`, of the surrogate class of the interface `I2`. The called access method is a delegation method, and thus the call is delegated to the dispatch method of the class `S1`. However, as the dispatch method is overridden by an access method generated from the `represents` clause of the class `S2`, the delegation results in calling the access method of the class `S2`, which returns an appropriate abstract value for the model field `val2`. The abstract value of the model field `val2` in turn becomes the abstract value of the model field `val1`, and as the result, the precondition of the method `m1` is evaluated with a proper value for the model field `val1`.

Another interesting case is the appearance of the interface model field `val1` in the precondition of the method `m2` in the class `S2`. An interface model field is used in an assertion of a class. However, the same mechanism works here. The precondition method of the method `m2` calls the access method of the interface model field `val2` added to the surrogate class of the interface `I1`, as that is where the model field is declared. The rest of access method call sequence are the same; i.e., the call sequence results in calling the model field access method for `val2` of the class `S2`, which in turns gives the abstract value for `val1`.

Note that all references to interface model fields are translated into calls to the corresponding delegation methods, which delegate the calls to the corresponding dispatch methods of the implementing classes. As an optimization, it is possible to eliminate these extra calls of delegation methods, e.g., by translating model field references directly to calls to the corresponding dispatch methods.

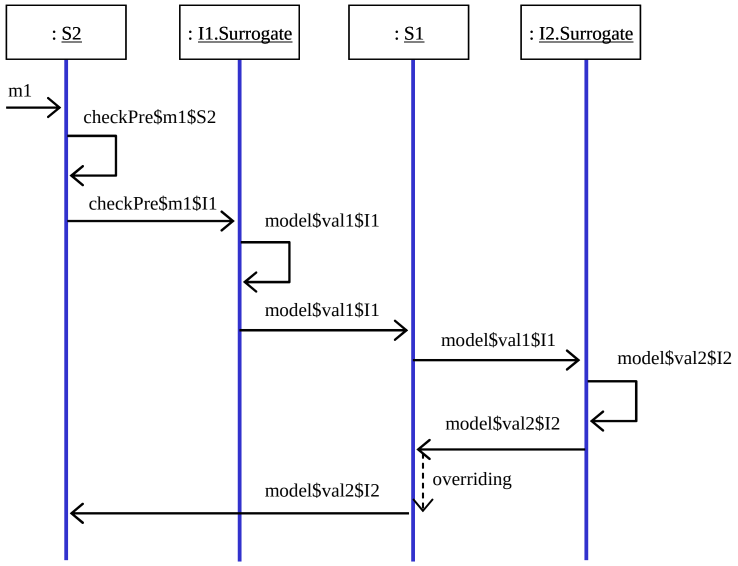

Figure 7.9: Message sequence diagram to check the method `m1`'s precondition.

7.4 Ghost Fields

A *ghost field* is a specification-purpose field similar to a model field [50] [89] [95]. Unlike a model field, however, the value of a ghost field is not implicitly mapped from the program state, but is explicitly set with the `set` specification statement. The `set` statement is similar to Java assignment statements.

The JML compiler's approach is to turn a ghost field into a private program field in the host class, and generates a pair of access methods, called a *ghost field getter method* and a *ghost field setter method*. The first is for reading the field and the second is for writing into it. Both methods have public-accessibility to allow dynamic calls using Java's reflection facility; the approach supports separate compilation. A reference to a ghost field is translated into a call to the corresponding getter access method, and a `set` statement is translated into a call to the corresponding setter access method. The approach allows to use the same infrastructure for both model and ghost fields. For example, given a ghost field "`ghost T v;`" declared in a type S, the JML compiler generates the code shown in Figure 7.10 and adds it to the host class[6].

Translating each ghost field reference into a call to the corresponding access method is the same as that of a model field (see Section 7.2). Consider an expression $E.v$, where $v$ is a ghost field of type T declared in a class S, and E is of type S (or its subclasses). Then, the expression is translated into $\mathcal{D}[\![E.\text{ghost\$}v\$S()]\!]$, where the notation $\mathcal{D}[\![E.m()]\!]$ means that the method m is dynamically invoked using Java's reflection facility[7]. A `set` statement is translated in a similar way (refer to Section 4.7.3 for the translation of `set` statements).

A ghost field can also be declared in an interface, and an interface ghost field is treated in a similar fashion. However, one importance difference is that the receiver of an access method call must be a surrogate object, as the accessor method is hosted by the surrogate class of the interface. As an example, suppose a ghost field access expression, $E.v$, where $v$ is a ghost field of type T declared in an interface I, and E is an expression of runtime type, say S, that implements the interface I.

[6] The `static`ness of a ghost field is carried over the access methods; i.e., if a ghost field is static, its access methods become static.

[7] As an optimization, reflective calls are used only if necessary; e.g., non-reflective calls are used for ghost fields declared in the same file.

```
// Generated from "ghost T v;"
private T ghost$v;

public T ghost$v$S() {
  return ghost$v;
}

public void ghost$v$S(T v) {
   ghost$v = v;
}
```

Figure 7.10: Ghost field access methods.

Then, the expression is translated into $[\![\mathcal{S}_{S,I}(E).\text{ghost\$}v\text{\$}I()]\!]$, where $\mathcal{S}_{S,I}$ is a function that maps each object of the class S into a unique surrogate object of the interface I (refer to Section 6.5 for surrogate classes).

It is crucial for each access method to have a unique name, e.g., by appending the owner type's name at the end, like `ghost$v$S`. This is because (ghost) fields are statically resolved while (access) methods are dynamically dispatched. For example, if both a subclass S and its superclass T declare the same ghost field, say $v$, then the one in S hides the one in T. However, it is still possible in S to refer to the hidden ghost field of T, e.g., by an expression `((T) this).v`, where the field $v$ is statically resolved by the compiler. If such an expression is translated into an expression of access method calls, e.g., `((T) this).v()`, the access method `v()` is dynamic dispatched. This means that the one in the subclass would be incorrectly invoked if both had the same name.

7.5 Model Methods

A *model method* is a specification-purpose method declared with the JML modifier `model`. It is normally declared to be pure, and is used in assertions. A model method can be either `static` or non-`static`, and be declared both in classes and interfaces. JML also supports a *model constructor*, which is a specification-purpose constructor.

Figure 7.11 shows an example of a model method. The class `TreeNode` defines a recursive data structure that may be used as nodes of binary trees. Each node has a value and both left and right nodes. To ease writing specifications in terms of the class `TreeNode`, a model field `values` is declared, that denotes the set of values contained in a tree rooted by a particular node. The abstraction function for the model field `values` is specified in the `represents` clause by using a model method, `abstractValue`. The model method calculates the values contained in a node by recursively collecting the values of its left and right subtrees. A model method is not required to have a body, though the example shows one that has. As shown, a model method is an excellent tool to specify the abstraction function for a model field.

The notion of model methods makes the role of each method clear, i.e., whether it is intended for use only in specifications or to be a part of the implementation. Without it, one has to write specification-purpose methods as Java program methods. Such methods may clutter program code and blur the boundary between specifications and programs, thereby confusing the reader about the intended use. As they are program methods, they are still compiled into bytecode even though not compiled with runtime assertion checks. On the contrary, model methods make their purposes clear and are not compiled into bytecode if not needed.

Two main challenges for model methods are executability and separate compilation. Not all model methods are executable; e.g., a model method may not have a body. Due to separate compilation, it is not known at compiled-time whether a model method would be compiled into bytecode or

```
//@ model import java.util.*;

public class TreeNode {
  private int value;
  private TreeNode left, right;

  //@ public model non_null Set values;
  //@ private represents values <- abstractValue();

  /*@ ensures \result != null;
    @ private model pure Set abstractValue()
    @   throws IllegalStateException {
    @   Set ret = new HashSet();
    @   ret.add(new Integer(value));
    @   if (left != null) {
    @     ret.addAll(left.abstractValue());
    @   }
    @   if (right != null) {
    @     ret.addAll(right.abstractValue());
    @   }
    @   return ret;
    @ }
    @*/

  // ...
}
```

Figure 7.11: Example of model methods.

not. The JML compiler's approach is to determine the executability of a model method at compile-time when generating an appropriate access method. In particular, a non-executable model method is translated into an access method that throws an angelic undefinedness exception; thus, all calls to such a model method are interpreted contextually by the parent expressions (see Section 3.2 for undefinedness and contextual interpretation). To support separate compilation, all model method calls are translated into dynamic calls unless the model methods are declared in the same file.

The executability of a model method depends on several factors. A model method is *executable* if it satisfies all the following conditions.

1. It is declared by an *executable* class or interface. A Java program class or interface is always executable, but a model class or interface is not executable yet.

2. All its signature types are executable classes or interfaces. A signature type of a method is the return type, one of its formal parameter types, or one of its exception types listed in the `throws` clause.

3. It has a body and the body is executable in that it consists of Java statements and JML specification statements such as `assert`, `assume`, and loop invariants and variants[8] (refer to Section 4.7.3 for specification statements).

[8] This condition is not yet checked by the JML compiler. It is left as a future work to support more features here, e.g., allowing in the body model fields, ghost fields, `spec_public` fields, `spec_protected` fields, and model methods.

An executable model method is translated into a Java program method by basically uncommenting the JML annotation markers. However, this involves some complication. A model method is turned into an access method of public visibility to allow dynamic calls using Java's reflection facility. Why an access method with a fresh name is generated instead of turning the model method into a Java program method? Such a program method must be public to allow dynamic calls, which is essential to support separate compilation. However, making it public produces unintended side-effects such as name clash, unintended inheritance, and method overriding, as a non-public method may now become public. So, access methods are used for model methods.

Once model methods are translated, then the next step is to translate their use in assertions. The goal here is to support separate compilation and also to make the translated code efficient in terms of runtime speed. For separate compilation, a model method call is translated into a dynamic method call to the corresponding access method. However, it is often possible to translate a model method call into a static method call. An example is the use of model method in the same file; all other calls are translated into dynamic calls. If a dynamic call fails at runtime, the call is interpreted as angelic undefinedness (refer to Section 3.2 for undefinedness of assertions).

A model method declared in an interface is treated similarly. An executable form of interface model method becomes an access method in the surrogate class of the interface. As before, a model method call in assertions is translated into either a static call or a dynamic call depending on the context. However, the translation becomes complicated, as the invoked method now would be defined in the surrogate class of the interface. The receiver for such a method call must be a surrogate class for a static method and a surrogate object for an instance method. As an example, consider a model method call expression, E_0.m(E_1,...,E_n), where m is a model method declared in an interface, I, and E_0 is an expression of runtime type, T, implementing the interface I. Then, the expression is translated into $T_I(E_0)$.m(E_1,...,E_n), where T_I is a function that maps each object of runtime type T into a unique surrogate object of the interface I. As a surrogate may have its own state, each instance of a class is associated with a unique surrogate object for each interface that the class implements (refer to Section 6.5 for surrogates).

A model constructor, that can be declared only in a class, is treated in a similar fashion. An executable model constructor becomes a private Java constructor, regardless of its visibility. In addition, a static factory method is added as an access method (refer to Gamma et al.'s book [52] for the factory design pattern). The use of model constructors is the same as that of Java program constructors. In Java, a class instance creation expression (e.g., `new TreeNode(10, null, null)`) is used to create a new instance of a class, and an applicable constructor of the class is chosen at compile-time [55, Section 19.5]. If the applicable constructor is a model constructor, then depending on the context the expression is translated into a dynamic or static call to the corresponding factory access method. This factory method calls the appropriate private constructor.

7.6 Discussion

The use of abstract, specification-only fields and methods can be seen as an evolutionary advance over the design-by-contract approaches [108] [109] [110] [111] that only use program fields and methods for writing assertions. The main distinction is that with such fields and methods, one knows exactly what parts of the code are only for specification purposes. That is, the notion of "modelness" in JML makes the role of each declaration clear both to the reader and the tools.

The approach discussed in this chapter has a few limitations and shortcomings. The approach cannot execute abstraction relations [6] [45] [92] [138] [140] specified in `represents` clauses for model fields. Given an abstraction relation for a model field, the approach cannot build (or retrieve) for the field an abstract value from the program state; all references to such a model field are treated as angelic undefinedness. This means that one has to specify an abstraction function to make a model field executable. Another shortcoming is runtime performance. In the approach, each reference to a model field results in the construction of a new abstract value. In the class `TreeNode`, (see Figure 7.11), for example, each reference to the model field `values` leads to a construction of a new `HashSet` that requires traversing the whole tree rooted by the node. A frequent construction of abstract values from the program state may affect the performance of runtime assertion checking (e.g., speed and heap memory), in particular, for large, container-style data structures. Dynamic calls may also affect the speed of runtime assertion checking. They support separate compilation, and make assertion checking faithful to the semantics of JML, but may decrease the runtime speed because of their reflective nature. It is a future research topic to measure precisely the impact of dynamic calls to the runtime speed of assertion checking code, and to completely eliminate the use of dynamic calls (see Section 9.1).

In addition to specification-purpose fields and methods, JML also allows specification-purpose classes and interfaces, called *model classes and interfaces*. However, model types are not yet supported by the JML compiler. All references to model types are currently interpreted angelically by their parent expressions.

In JML, one can change the visibility of a Java declaration; it can be widened from private or package-visible to protected or public, by using the JML modifiers `spec_protected` or `spec_public` [89]. Such a declaration is treated in the JML specification scope as if it is declared as protected or public. For example, a spec_protected, private field is treated like a protected field in the JML specification scope. It can appear in a protected specification, and is inherited to the specifications of subclasses. The JML compiler supports `spec_protected`ness and `spec_public`ness of fields and methods (including constructors). The approach is similar to those of specification-purpose fields and methods; it is based on both access methods and dynamic calls. An appropriate access method is generated for such a field, and all references to it are translated into dynamic calls to the access method. For a constructor, the access method becomes a static factory method, that implicitly invokes the corresponding constructor by having an appropriate class instance creation expression and returns a new instance of the class. Of course, the factory method becomes public to allow dynamic calls for separate compilation.

Chapter 8

Application — Unit Testing with JML

Writing unit test code is labor-intensive, hence it is often not done as an integral part of programming. However, unit testing is a practical approach to increasing the correctness and quality of software; for example, the Extreme Programming approach [7] relies on frequent unit testing.

This chapter demonstrates how the runtime assertion checker can help automate unit testing, by presenting a new approach that makes writing unit tests easier. The approach uses a formal BISL's runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles. These oracles can be easily combined with hand-written test data. Instead of writing testing code, the programmer writes formal specifications (e.g., pre- and post-conditions). This makes the programmer's task easier, because specifications are more concise and abstract than the equivalent test code, and hence more readable and maintainable. Furthermore, by using specifications in testing, specification errors are quickly discovered, so the specifications are more likely to provide useful documentation and inputs to other tools. The approach is implemented by using JML and the JUnit testing framework, but is applicable to other combinations of formal BISLs and unit test tools. This chapter is adapted from my early work [29].

8.1 Introduction

Program testing is an effective and practical way of improving correctness of software, and thereby improving software quality. It has many benefits when compared to more rigorous methods like formal reasoning and proof, such as simplicity, practicality, cost effectiveness, immediate feedback, understandability, and so on. There is a growing interest in applying program testing to the development process, as reflected by the Extreme Programming (XP) approach [7]. In XP, unit tests are viewed as an integral part of programming. Tests are created before, during, and after the code is written — often emphasized as "code a little, test a little, code a little, and test a little ..." [8]. The philosophy behind this is to use regression tests [80] as a practical means of supporting refactoring.

8.1.1 The Problem

However, writing unit tests is a laborious, tedious, cumbersome, and often difficult task. If the testing code is written at a low level of abstraction, it may be tedious and time-consuming to change it to match changes in the code. One problem is that there may be too much testing code that has to be examined and revised. Another problem occurs if the testing program refers to details of the representation of an abstract data type; in this case, changing the representation may require changing the testing program.

To avoid these problems, one should automate more of the writing of unit test code. The goal is to make writing testing code easier and more maintainable. One way to do this is to use a framework that automates some of the details of running tests. An example of such a framework is JUnit [8] [70]. JUnit is a simple yet practical testing framework for Java classes; it encourages the close integration of testing with development by allowing a test suite be built incrementally. However, even with tools like JUnit, writing unit tests often requires a great deal of effort. Separate testing code must be written and maintained in synchrony with the code under development, because the test class must inherit from the JUnit framework. This test class must be reviewed when the code under test changes, and, if necessary, also revised to reflect the changes. In addition, the test class suffers from the problems described above. The difficulty and expense of writing the test class are exacerbated during development, when the code being tested changes frequently. As a consequence, during development there is pressure to not write testing code and to not test as frequently as might be optimal.

We encountered these problems ourselves in the JML Project. We have been formally documenting in JML the behavior of some implementation classes of JML tools. This enabled us to use JML's runtime assertion checker to help debug our code. In addition, we have been using JUnit as our testing framework. We soon realized that we spent a lot of time writing test classes and maintaining them. In particular we had to write many query methods to determine test success or failure. We often also had to write code to build expected results for test cases. We also found that refactoring made testing painful; we had to change the test classes to reflect changes in the refactored code. Changing the representation data structures for classes also required us to rewrite code that calculated expected results for test cases.

While writing unit test methods, we soon realized that most often we were translating pre- and postconditions into the code in corresponding testing methods. The preconditions became the criteria for selecting test inputs, and the postconditions provided the properties to check for test results. That is, we turned the postconditions of methods into code for test oracles. A *test oracle* determines whether or not the results of a test execution are correct [123] [132] [144]. Developing test oracles from postconditions approach helped avoid dependence of the testing code on the representation data structures, but still required us to write lots of query methods. In addition, there was no direct connection between the specifications and the test oracles, hence they could easily become inconsistent.

These problems led us to think about ways that would save us time and effort in testing code. We also wanted to have less duplication of effort between the specifications we were writing and the testing code. Finally, we wanted the process to help keep specifications, code, and tests consistent with each other.

8.1.2 Approach

As a solution to these problem, a simple and effective approach is proposed to automate the generation of oracles for unit testing. The conventional way of implementing a test oracle is to compare the test output to some pre-calculated, presumably correct, output [60] [119]. The approach takes a different perspective. Instead of building expected outputs and comparing them to the test outputs, the specified behavior of the method under test is monitored to decide whether a test passed or failed. This monitoring is done by using the formal BISL's runtime assertion checker. The approach thus combines formal specifications (such as JML) and a unit testing framework (such as JUnit).

Formal interface specifications include class invariants and pre- and postconditions. These specifications are assumed to be fairly complete descriptions of the desired behavior. Although the testing process will encourage the user to write better preconditions, the quality of the generated test oracles will depend on the quality of the specification's postconditions. The quality of these postconditions

```
public class Person {
  private /*@ spec_public non_null @*/ String name;
  private /*@ spec_public @*/ int weight = 0;
  //@ public invariant weight >= 0;

  /*@ public normal_behavior
    @   assignable name, weight;
    @   ensures n.equals(name) && weight == 0; @*/
  public Person(/*@ non_null @*/ String n) { name = n; }

  /*@ public behavior
    @   assignable weight;
    @   ensures kgs >= 0 && weight == \old(weight + kgs);
    @   signals (IllegalArgumentException e) kgs < 0; @*/
  public void addKgs(int kgs) { weight += kgs; }

  /*@ public normal_behavior
    @   ensures \result == weight; @*/
  public /*@ pure @*/ int getWeight() { return weight; }

  // ...
}
```

Figure 8.1: Example JML specification. The method `addKgs` contains an error to be revealed in Section 8.5.2.

is the user's responsibility, just as the quality of hand-written test oracles would be.

A tool was written to generate JUnit test classes from JML specifications. The generated test classes send messages to objects of the Java classes under test; they catch assertion violation errors from test cases that pass an initial precondition check. Such assertion violation errors are used to decide if the code failed to meet its specification, and hence that the test failed. If the class under test satisfies its interface specification for some particular input values, no such errors will be thrown, and that particular test execution succeeds. So the generated test code serves as a test oracle whose behavior is derived from the specified behavior of the target class. (There is one complication which is explained in Section 8.4.) The user is still responsible for generating test data; however the generated test classes make it easy for the user to add test data.

8.1.3 Outline

The remainder of this chapter is organized as follows. Section 8.2 describes the capabilities that the approach assumes from a formal BISL and its runtime assertion checker, using JML as an example. Section 8.3 describes the capabilities that the approach assumes from a testing framework, using JUnit as an example. Section 8.4 explains the approach in detail, including deciding test success or failure, setting up test fixture, and automatic generation of test classes. Section 8.5 explains how the user adds test data to the generated test classes, and how to run actual tests. Section 8.6 discusses related work, and in Section 8.7 I summarize this chapter.

8.2 Assumptions About the Specification Language

The approach assumes that the formal BISL specifies the interface and behavior of classes and methods. The language is assumed to have a way to express class invariants and method specifications consisting of pre- and postconditions.

The approach can also handle specification of some more advanced features. One such feature is *in-line assertions*, such as `assert` statements, `assume` statements, and loop invariants and variants (see Section 4.7). Another feature is a distinction between normal and exceptional postconditions. A *normal postcondition* describes the behavior of a method when it returns without throwing an exception; an *exceptional postcondition* describes the behavior of a method when it throws an exception.

JML is an example of such a formal BISL [88] [89] (see also Chapter 2). Figure 8.1 shows a JML specification that is used as a running example in this chapter.

8.2.1 The Runtime Assertion Checker

The specification language is assumed to have a runtime assertion checker. The basic task of the runtime assertion checker is to execute code in a way that is transparent, unless an assertion violation is detected. That is, if a method is called and no assertion violations occur, then, except for performance measures (e.g., time and space) the behavior of the method is unchanged [48]. In particular, this implies that, as in JML, assertions can be executed without side effects [90].

The runtime assertion checker is not assumed to execute all assertions in the specification language. However, only the assertions it can execute are of interest.

The runtime assertion checker is assumed to have a way of signaling assertion violations to a method's callers. In practice this is most conveniently done using exceptions. While any systematic mechanism for indicating assertion violations would do, to avoid circumlocutions, it is assumed that exceptions are used in the remainder of this chapter.

The runtime assertion checker must have some exceptions that it can use without interference from user programs. These exceptions are thus reserved for use by the runtime assertion checker. Such exceptions are called assertion violation exceptions. It is convenient to assume that all such assertion violation exceptions are subtypes of a single assertion violation exception type.

The approach assumes that the runtime assertion checker can distinguish two kinds of precondition assertion violations: *entry precondition violations* and *internal precondition violations*. The former refers to violations of preconditions of the method being tested. The latter refers to precondition violations that arise during the execution of the tested method's body. Other distinctions among assertion violations are useful in reporting errors to the user, but are not important for the approach.

JML's runtime assertion checker can execute a constructive subset of JML assertions, including some forms of quantifiers. In functionality, it is similar to other design by contract tools [81] [109] [110] [134]; such tools could also be used with the approach. In JML, the assertion violation exceptions are organized into an exception hierarchy (refer to Section 4.1.1). The ultimate superclass of all assertion violation exceptions is the abstract class `JMLAssertionError`. This class has several subclasses that correspond to different kinds of assertion violations, such as precondition violations, postcondition violations, invariant violations, and so on. The entry precondition violation and the internal precondition violation of the assumptions correspond to the types `JMLEntryPreconditionError` and `JMLInternalPreconditionError`. Both are concrete subclasses of the abstract class `JMLPreconditionError`.

8.3 Assumptions About the Testing Framework

The approach assumes that unit tests are to be run for each method of the class being tested. The framework is assumed to provide *test methods*, which are methods used to test the methods of the class under test. For convenience, test methods are assumed to be grouped into test classes.

In the approach, each test method executes several test cases for the method it is testing. Thus it is assumed that a test method can indicate to the framework whether each test case fails, succeeds, or is meaningless. The outcome will be meaningless if an entry precondition violation exception occurs for the test case; details are given in Section 8.4.1.

It is also assumed that there is a way to provide test data to test methods. Following JUnit's terminology, this is called a test fixture. A *test fixture* is a context for executing a test; it typically contains several declarations for test inputs and expected outputs. For the convenience of the users, assumed is a global test fixture that is shared by all test methods in a test class. With a global test fixture, one needs ways to initialize the test inputs, and to undo any side effects of a test after running the test.

JUnit is a simple, useful testing framework for Java [8] [70]. In JUnit, a test class consists of a set of test methods. The simplest way to tell the framework about the test methods is to name them all with names beginning with "`test`". The framework uses introspection to find all these methods, and can run them when requested.

Figure 8.2 is a sample JUnit test class, which is designed to test the class `Person`. Every JUnit test class must be a subclass, directly or indirectly, of the framework class `TestCase`. The class `TestCase` provides a basic facility to write test classes, e.g., defining test data, asserting test success or failure, and composing test methods into a test suite.

One uses methods like `assertEquals`, defined in the framework, to write test methods, as in the test method `testAddKgs`. Such methods indicate test success or failure to the framework. For example, when the arguments to `assertEquals` are not equal, the test fails. Another such framework method is `fail`, which directly indicates test failure. JUnit assumes that a test succeeds unless the test method throws an exception or indicates test failure. Thus the only way a test method can indicate success is to return normally.

JUnit thus does not provide a way to indicate that a test execution is meaningless. This is because it is geared toward counting executions of test methods instead of test cases, and because hand-written tests are assumed to be meaningful. Thus, it is necessary to extend JUnit to count test cases and to track which ones are meaningful. This is done by providing a new JUnit framework class, `JMLTestRunner`, which tracks the meaningful test cases executed.

JUnit provides two methods to manipulate the test fixture: `setUp` creates objects and does any other tasks needed to run a test, and `tearDown` undoes otherwise permanent side-effects of tests. For example, the `setUp` method in Figure 8.2 creates a new `Person` object, and assigns it to the test fixture variable `p`. The `tearDown` method can be omitted if it does nothing. JUnit automatically invokes the `setUp` and `tearDown` methods before and after each test method is executed (respectively).

The static method `suite` creates a *test suite*, i.e., a collection of test methods. To run tests, JUnit first obtains a test suite by invoking the method `suite`, and then runs each test method in the suite. A test suite can contain several test methods, and it can contain other test suites, recursively. Figure 8.2 uses Java's reflection facility to create a test suite consisting of all the test methods of class `PersonTest`.

8.4 Test Oracle Generation

This section presents the details of generating JUnit test classes from JML-annotated Java classes and interfaces. It first describe how test outcomes are determined and how the test classes are structured, and then discuss the actual generation of test methods and classes.

```
import junit.framework.*;

public class PersonTest extends TestCase {
  private Person p;

  public PersonTest(String name) {
    super(name);
  }

  public void testAddKgs() {
    p.addKgs(10);
    assertEquals(10, p.getWeight());
  }

  protected void setUp() {
    p = new Person("Yoonsik");
  }

  public static Test suite() {
    return new TestSuite(PersonTest.class);
  }

  public static void main(String args[]) {
    String[] testCaseName = {PersonTest.class.getName()};
    junit.textui.TestRunner.main(testCaseName);
  }
}
```

Figure 8.2: Sample JUnit test class for the class `Person`.

8.4.1 Deciding Test Outcomes

There is a separate test method, `test`M, for each method, m, to be tested, and the test method `test`M runs m on several test cases. A *test case*, $(o, \vec{x})$, is a pair consisting of a receiver, o, and a sequence of arguments, $\vec{x}$; of course, test cases don not include the receivers for static methods. The question is how the method `test`M decides its outcome for a given test case $(o, \vec{x})$.

For a given test case $(o, \vec{x})$, the test method `test`M monitors the call to the method m (i.e., $o.m(\vec{x})$) to decide the test outcome. If the call terminates normally, i.e., no exception is thrown, then the test succeeds; the method m must have satisfied its specification for the call, because there was no assertion violation.

Similarly, if the call to m results in an exception that is not an assertion violation error, then the test succeeds. As runtime assertion checking is transparent, such an exception must have been passed along by the runtime assertion checker (see Section 4.5.2). This means that the method m must have satisfied its specification for the call, specifically, the exceptional postcondition. With JUnit, however, such an exception must be caught by the test method `test`M, because all exceptions are interpreted by the JUnit framework as signaling test failures. In sum, the method `test`M must catch and ignore all exceptions that are not assertion violation errors.

If the call to m throws an assertion violation error, things become interesting. If the assertion violation error is not a precondition error, then the method m is considered to fail that test case. However, one has to be careful with the treatment of precondition violations. A precondition is an obligation that the client must satisfy; nothing else in the specification is guaranteed if the precondition is violated. Therefore, when the test method `test`M calls the method m and m's precondition does not hold, it is not considered to be a test failure; rather, it indicates that the given test case is outside m's domain, and thus is inappropriate for test execution. The outcome of such a test execution is called *meaningless* instead of being either a success or a failure. On the other hand, precondition violations that arise inside the execution of m should still be considered to be test failures. To do this, distinguished are two kinds of precondition violations that may occur when `test`M runs m on a test case, $(o, \vec{x})$:

- The precondition of m fails for $(o, \vec{x})$, which indicates, as above, that the test case $(o, \vec{x})$ is outside m's domain. As noted earlier, this is called an *entry* precondition violation.

- A method f called from within m's body signals a precondition violation, which indicates that m's body did not meet f's precondition, and thus that m failed to correctly implement its specification on the test case $(o, \vec{x})$. (Note that if m calls itself recursively, then f may be the same as m.) Such an assertion violation is an *internal* precondition violation.

When the JML runtime assertion checker detects a precondition violation of the second kind, it converts the precondition violation into an internal precondition violation error (see Section 4.3). Thus, `test`M decides that m fails on a test case $(o, \vec{x})$ if m throws an internal precondition violation error, but rejects the test case $(o, \vec{x})$ as meaningless if it throws an entry precondition violation error. This treatment of precondition error is the main change that was made to JML's runtime assertion checker. The treatment of meaningless test case executions is also the only extension made to the JUnit framework.

What should be the outcome if a test case encounters an invariant violation at the pre-state? Such a situation can arise if clients can directly write an object's fields, or if aliasing allows clients to manipulate the object's representation without calling its methods. The question is whether such invariant violations should be treated as a test failure or as a rejection of the test data (i.e., as a meaningless test). One reason for rejecting the test data is that one can consider the invariant to be part of the precondition. One may also consider an object malformed if it does not satisfy the invariant. However, treating such violations as if the test case were meaningless may mask the underlying violation of information hiding, and so the current implementation treats them as test failures.

To summarize, the outcome of a test execution is "failure" if an assertion violation error other than an entry precondition violation is thrown, is "meaningless" if an entry precondition violation is thrown, and "success" otherwise.

8.4.2 Setting Up Test Cases

A test fixture is responsible for constructing test cases such as receivers and arguments. For example, testing the method `addKgs` of class `Person` (see Figure 8.1) requires a receiver of type `Person` and an argument of type `int`. There is no need to construct expected outputs, because test results are determined by observing the runtime assertion checker, not by comparing actual outputs to the expected.

How does the user define test cases for use with the generated test methods? There are several possibilities:

- *Separate test fixture.* Each test method has a separate set of test fixture variables, resulting in a very flexible and customizable configuration. However, defining such fixture variables becomes complicated and requires more work from the user.

- *Global test fixture.* All test methods share the same set of test fixture variables. The approach is simple and intuitive, and thus defining fixture variables requires less work from the user. However, the user has less control in that, because of shared fixture variables, it becomes hard to supply specific test cases to specific test methods.

- *Combination.* This combines the above two approaches to have a simple and intuitive test fixture configuration, and yet to give more control to the user.

An earlier work [29] adopted the global test fixture approach. The rationale was that the more test cases would be the better and the simplicity of use would outweigh the benefit of more control. There would be no harm to run test methods with test cases of other methods (if test cases are type-compatible). Some of test cases might violate the precondition; however, entry precondition violations are not treated as test failures, and so such test cases cause no problems. However, the experience showed that the approach often leads a dramatic increase in testing time; all combinations of the receivers and arguments are tested with fixture variables reinitialized for teach test execution, and also lots of unneeded initializations happen. The combination approach is used in the current implementation.

In the combination approach, a test fixture variable is introduced for each formal parameter type. In addition, a separate fixture variable is introduced for the receivers. Fixture variables are defined as arrays. The idea is to provide a vector of values for the receiver and each argument. As in the global fixture approach, the fixture variables are shared by all test methods; all test method shares the receiver fixture variable, and if more than one method have the same parameter type, they share the same fixture variable for that argument. For example, the class `Person` will have the following test fixture variables; its methods have only two formal parameter types, `String` in the constructor and `int` in the method `addKgs`.

```
protected Person[] receivers;
protected String[] vString;
protected int[] vint;
```

The first fixture variable named `receivers` is for receivers, and the others are for arguments. A simple naming convention is adopted for test fixture variables so that they can be shared by all test methods (see Section 8.4.4). A fixture variable's name is the name of its type prefixed by the character `v`[1], e.g., `vint` for type `int`. The test fixture variables become `protected` fields of the test class so that the user can initialize them in subclasses (see Section 8.5.1 for details).

The fixture variables for a particular test method are those that correspond to the method's parameter types plus the receiver, and test cases for the method are all possible combinations of the method's fixture variables. For example, the set of test cases for the method `addKgs` is defined as follows.

$$\{\langle \texttt{receivers[}i\texttt{]}, \texttt{vint[}j\texttt{]} \rangle \mid 0 \leq i < \texttt{receivers.length},\ 0 \leq j < \texttt{vint.length}\}$$

In addition to the test fixture variables, the combination approach generates a pair of initialization and uninitialziation methods for each test fixture variable[2] (see Figure 8.3). The user uses these fixture methods to initialize and uninitialize test fixture variables, thus supplying test cases to test methods (see Section 8.5.1 for details). The name of method under test is passed as an argument so that the user can have more control over test cases. A test method calls the initialization and uninitialziation methods of its test fixture variables before and after each test execution (see Section 8.4.4 for more details).

8.4.3 Test Methods

Recall that there will be a separate test method, `test`M for each method, m, to be tested. The purpose of `test`M is to determine the outcome of calling m with each test case and to give an

[1] For an array type, the character `$` is used to denote its dimension, e.g., `vint_$_$` for `int[][]`.

[2] I thank Professor Gary T. Leavens for implementing this extension to the earlier global test fixture approach [29]. I also thank David Cok for pointing out problems with the earlier approach and discussing such an extension.

```
protected abstract void init_receivers(String forMethodName);
protected void uninit_receivers(String forMethodName) {}
protected abstract void init_vString(String forMethodName);
protected void uninit_vString(String forMethodName) {}
protected abstract void init_vint(String forMethodName);
protected void uninit_vint(String forMethodName) {}
```

Figure 8.3: Test fixture methods for the class `Person`.

informative message if the test execution fails for that test case. The method `test`M accomplishes this by invoking m with each test case and indicating test failure when the runtime assertion checker throws an assertion violation error that is not an entry precondition violation. Test methods also note when test cases were rejected as meaningless.

Figure 8.4 shows an example of generated test methods, the test method for `addKgs` of the class `Person`. A simple convention is used to name test methods; the original method name is prefixed with the string "`test`" and capitalize the first letter of the method name.[3] The method tests all combinations test fixture variables for the method under test, with nested loops over the receiver and the arguments. The corresponding fixture variable is initialized before and uninitialized after each loop by calling the corresponding fixture methods, e.g., `init_receivers` and `uninit_receivers`. The reinitialization of fixture variables prevents side-effects of one test execution, e.g., mutation of the receiver or argument objects, from being carried over to another. To improve readability, the test method introduces local variables for the test fixture variables corresponding to the formal parameters of the method under test. The local variables are named the same as the formal parameters of the method under test. For each test case, given by the test fixture variables, the test method then invokes the method under test in a `try` statement and checks if the runtime assertion checker detects an assertion violation. As described above, an assertion violation error (`JMLAssertionError`) other than an entry precondition violation error means a failure of the test execution; thus an appropriate error message is printed. The method also keeps track of test statistic, e.g., the ratio of meaningless test cases.

A similar form of test methods is generated for each static method and constructor. For them, however, the outermost loop is omitted, as test messages are sent to the class object itself or new instances are created to monitor runtime assertion checking.

8.4.4 Test Classes

For each class to be tested, say, C, a separate JUnit test class, C`_JML_Test`, is generated, which contains test fixture variables and test methods described in the previous section; e.g., for the class `Person`, a test class `Person_JML_Test` is generated. The test class also includes several boilerplate methods such as constructors, main methods, and test suite methods. As required by JUnit, the test class is defined to be a subclass of the framework class `TestCase`. The `package` and `import` declarations are verbatim copied from C to C`_JML_Test`. As a result, the test class will reside in the same package as the class to be tested; this allows one to test package-visible methods. In addition, several new `import` declarations are added to import JUnit-specific packages.

[3] If necessary, a unique suffix is appended to prevent a name clash due to method overloading.

```
public void testAddKgs() {
  init_receivers("addKgs");
  for (int i = 0; i < receivers.length; i++) {
    init_vint("addKgs");
    final int[] kgs = this.vint;
    for (int j = 0; j < kgs.length; j++) {
      try {
        if (receivers[i] == null) {
           /* ... tell framework test case was meaningless ... */
           continue;
        }
        receivers[i].addKgs(kgs[j]);
      }
      catch (JMLEntryPreconditionError e) {
        /* ... tell framework test case was meaningless ... */
        continue;
      }
      catch (JMLAssertionError e) {
        String msg = /* a String showing the test case */;
        fail(msg + NEW_LINE + e.getMessage());
      }
      catch (java.lang.Throwable e) {
        continue;
      }
      finally {
        /* ... accumulate testing statistic ... */
      }
      uninit_vint("addKgs");
    }
    uninit_receivers("addKgs");
  }
}
```

Figure 8.4: Generated test method for the method `addKgs` of the class `Person`. Suppressed are details of generating error messages and telling the framework about meaningless test cases.

8.5 Test Execution

8.5.1 Supplying Test Cases

The user has to supply test data to test classes by initializing their test fixture variables. This is done by defining subclasses, called *test case classes*, of the test classes and by overriding the inherited test fixture methods. Remember that a pair of initialization and uninitialization method is added for each test fixture variable (see Section 8.4.2). Organizing test data as subclasses of test classes allows one to regenerate test classes without losing existing test data. The skeleton code for test case classes can also be generated by the tool (see Section 8.5.2). The convention is to use the postfix `_JML_TestCase` for test case classes, e.g., `Person_JML_TestCase` for the class `Person`.

Figure 8.5 shows a user-defined test case class for the class `Person`. The user modified a skeleton test case class by editing test fixture initialization methods such as `init_receivers`, `init_vString`, and `init_vint`, corresponding to three test fixture variables `receivers`, `vString`, and `vint`. Each fixture method is responsible for initializing the corresponding test fixture variable; e.g., the method

```
import junit.framework.*;
import junit.extensions.*;

public class Person_JML_TestCase extends Person_JML_Test {
  public Person_JML_TestCase(String name) {
    super(name);
  }

  public static void main(String[] args) {
    org.jmlspecs.jmlunit.JMLTestRunner.run(suite());
  }

  public static junit.framework.Test suite() {
    return new junit.framework.TestSuite(Person_JML_TestCase.class);
  }

  protected void init_receivers(String methodName) {
    receivers = new Person[] {
      new Person("Yoonsik"),
      new Person("Mihee")
    };
  }

  protected void init_vString(String methodName) {
     if (vString == null) {
        vString = new String[] { "Yoonsik" };
     }
  }

  protected void init_vint(String methodName) {
    if (vint == null) {
       vint = new int[] { 10, -22, 0, 1, 55, 3000, };
    }
  }
}
```

Figure 8.5: User-defined test case class for testing the class `Person`.

`init_receiver` initializes the fixture variable `receiver`. The name of the method under test is given as an argument. Thus, the fixture method may provide test cases specifically for certain methods. As shown in the example, test fixture variables for immutable type (e.g., `String`) and primitive types (e.g., `int`) may be initialized just once as an optimization. With the example configuration of test fixture variables, the `addKgs` method of the class `Person` is tested 12 times, one for each pair of `receivers[`i`]` and `vint[`j`]`, where $0 \leq i < 2$ and $0 \leq j < 6$.

In addition to supplying test data for generated test methods, one can also tune the testing by adding hand-written test methods to test case classes. The JUnit framework collects and exercises the added test methods together with the automatically generated methods. Thus, the approach allows one to combine automated and hand-written testing.

8.5.2 Running the Tests

Running JML test case classes (e.g., class Person_JML_TestCase) is the same as running JUnit test classes, as the test case classes are JUnit test classes. However, the class under test (e.g., class Person) must be compiled with runtime assertion checks[4]; otherwise, all tests would succeed, as there would be no runtime assertion violations. In general, running JML tests can be done in the following three steps.

1. Compile the class to be tested (e.g., class Person) with the JML compiler (jmlc).
2. Generate and compile a test class (e.g., class Person_JML_Test) with the tool jmlunit and a Java compiler such as javac.
3. Compile and run a test case class (e.g., class Person_JML_TestCase) with a Java compiler and an interpreter such as javac and java. A skeleton test case class can be generated by the tool jmlunit.

The first two steps can also be done by using the script jtest; for example, the command "jtest Person.java" does them assuming that the class Person is stored in the file Person.java.

The results of JML-based tests are presented to the user in the same way as JUnit tests are presented by the JUnit framework. Figure 8.6, for example, shows the result of running the test case class Person_JML_TestCase shown in Figure 8.5. One interesting aspect is the test statistic produced by JUnit. In JUnit, the number of test successes and failures (i.e., test runs) is counted in terms of test methods. However, this is not a right measure for JML-based tests, as each test method may run more than one test case, i.e., it tests all possible combinations of test data (see Section 8.4.3). Thus, a specialized test runner class, JMLTestRunner, is provided to have more accurate statistic on test runs. The test runner can be used in place of JUnit's test runner classes such as junit.framework.textui.TestRunner (see Figure 8.5). The class JMLTestRunner reports both meaningful and total numbers of test runs in terms of test cases, as shown in the last line of Figure 8.6. Such a report also prevents the user having a wrong impression that the class under test satisfied all tests when in fact no test has actually be executed due to all test cases being meaningless.[5] In addition, it can report coverage information, identifying assertions that are always true or always false, and thus indicating deficiencies in the set of test cases[6].

The example test output reveals the error that was mentioned earlier. in the caption of Figure 8.1. The error is shown by the test failure occurred while testing the method addKgs. In the output, the test data that caused the failure is printed, i.e., the receiver, an object of class Person with name Yoonsik, and the argument value -22. In addition, the framework also prints detailed information about the assertion violation encountered, including the location of the violated assertion and the objects and values involved.

A corrected implementation of the method addKgs is shown in Figure 8.7. The fix is to throw an appropriate exception when the argument is negative. Compare this with the specification and the faulty implementation shown in Figure 8.1.

8.6 Related Work

Several researchers have already noticed that if a program is formally specified, it should be possible to use the specification as an oracle [2] [123]. Thus, the idea of automatically generating test oracles

[4] Actually, the tool generates a separate test method that checks whether the class under test was compiled with the JML compiler or not. In addition, it also generates a separate test method that checks for the initialization of fixture variables.

[5] I thank an anonymous referee for pointing out this problem in my earlier work [29].

[6] I thank David Cok for implementing this feature.

```
....F..
Time: 0.12
There was 1 failure:
1) testAddKgs(Person_JML_TestCase)junit.framework.AssertionFailedError:
        Method 'addKgs' applied to
        Receiver receivers[0]: Person("Yoonsik",-22)
        Argument 'kgs' (vint[1]): -22
        Caused org.jmlspecs.jmlrac.runtime.JMLPostconditionError
        Assertion of method 'addKgs' of class 'Person' specified at
        Person.java:16:25 when
        'rac$old0' is -12
        'kgs' is -22
        'this' is Person("Yoonsik",-12)
        at Person_JML_Test.testAddKgs(Person_JML_Test.java:199)
        ...

FAILURES!!!
Tests run: 6,  Failures: 1,  Errors: 0
JML Tests run: 11/12 (meaningful/total)
```

Figure 8.6: Output from running the tests in `Person_JML_TestCase`.

```
public void addKgs(int kgs) {
    if (kgs >= 0)
       weight += kgs;
    else
       throw new IllegalArgumentException("Negative Kgs");
}
```

Figure 8.7: Corrected implementation of the method `addKgs`.

from formal specifications is not new, but the novelty lies in employing a runtime assertion checker as the test oracle engine. This aspect seems to be original and first explored in this chapter. Peters and Parnas's work also generates test oracles from formal program specifications [123]. The behavior of program is specified in a relational program specification using tabular expressions [64] [65], and the test oracle procedure, generated in C and C++, checks if an input and output pair satisfies the relation described by the specification. Their approach is limited to checking only pre and postconditions, thus allowing only black-box tests. The JML approach supports a form of white-box tests, as one can write in-line assertions that can be specified and checked within a method, i.e., on internal states (see Section 4.7). JML also supports abstract value manipulation and object-oriented concepts such as specification inheritance.

Antoy and Hamlet describe an approach to check the execution of an abstract data type's implementation against its specification [2]. Their approach is similar to the technique of multiversion programming [77] except that one version is an algebraic specification, serving as a test oracle. The algebraic specification is executed by (term) rewriting, and is compared with the execution of the implementation. For the comparison, the user has to provide an abstraction function that maps implementation states to abstract values. In JML, no separate (specification) program needs to run in parallel with the implementation. Interestingly, however, JML can simulate their approach to some extent by using ghost fields and the `set` specification statement (see Section 4.7.3 and 7.4).

The traditional way to implement test oracles is to compare the result of a test execution with a user supplied, expected result [60] [119]. A test case, therefore, consists of a pairs of input and output values. In the JML approach, however, a test case consists of only input values. And instead of directly comparing the actual and expected results, it is observed if, for the given input values, the program under test satisfies the specified behavior. As a consequence, programmers are freed from not only the burden of writing test programs, often called *test drivers*, but also from the burden of pre-calculating presumably correct outputs and comparing them. The traditional schemes are constructive and direct whereas the JML approach is behavior observing and indirect.

There are now quite a few runtime assertion checking facilities developed and advocated by many researchers. The earliest work is Meyer's design-by-contact implemented in the programming language Eiffel [108] [109] [110]. Eiffel's success in checking pre- and postconditions contributed to the availability of similar facilities in other programming languages, including C [134], C++ [42] [58] [129] [158], Java [5] [44] [71] [81] [120], .NET [3], Python [127], and Smalltalk [20]. The approaches vary widely from a simple assertion macros to built-in assertions and full-fledged contract enforcement. However, none is known to use its assertion checking capability as a basis for automated program testing. Thus, the JML approach is unique in the design-by-contract community in using a runtime assertion checking to automate program testing.

The above mentioned design-by-contract tools work only with concrete program values. However, in JML, one can specify behavior in terms of abstract (specification) values, rather than concrete program. values [88] [89]. JML's runtime assertion checker can evaluate a significant portion of abstract specifications — specifications written in terms of specification-purpose declarations such as model fields, ghost fields, and model methods (see Chapter 7).

There are many research papers published on the subject of testing using formal specifications [12] [24] [34] [80] [132]. Most are concerned with techniques and methods for automatically generating test cases from formal specifications, though there are some addressing the problem of automatic generation of test oracles as noted before [2] [123] [132]. A general approach is to derive the so-called *test conditions*, a description of test cases, from the formal specification of each program module [24]. The derived test conditions can be used to guide test selection and to measure comprehensiveness of an existing test suite, and sometimes they even can be turned into executable forms [24] [34]. The degree of support for automation varies widely from the derivation of test cases, to the actual test execution and even to the analysis of test results [34] [132]. Some approaches use existing specification languages [16] [61]. and others have their own (specialized) languages for the description of test cases and test execution [24] [34] [76] [132] [136]. All of these works are complementary to the approach described here, since, except as noted above, they solve the problem of defining test cases which I do not attempt to solve, and they do not solve the problem of easing the task of writing test oracles, which I partially solve.

8.7 Summary

I have presented a simple but effective approach to implementing test oracles from formal behavioral interface specifications. The idea is to use the runtime assertion checker as the decision procedure for test oracles. I have implemented this approach using JML, but other runtime assertion checkers can be adapted to work with the approach. However, there are two complications. The first is that the runtime assertion checker has to distinguish two kinds of precondition violations: those that arise from the call to a method and those that arise within the implementation of the method; the first kind of precondition violations is used to reject meaningless test cases, while the second indicates a test failure. The second is that the unit testing framework needs to distinguish three possible outcomes for test cases: a test execution can either be a success, a failure, or it can be meaningless.

The approach trades the effort one might spend in writing code to construct expected test outputs and test drivers for effort spent in writing formal specifications. Formal specifications are more concise and abstract than code, and hence I expect them to be more readable and maintainable.

Formal specifications also serve as more readable documentation than testing code, and can be used as input to other tools such as extended static checkers [37].

Most testing methods do not check behavioral results, but focus only on defining what to test. Because most testing requires a large number of test cases, manually checking test results severely hampers its effectiveness, and makes repeated and frequent testing impractical. To remedy this, my approach automatically generates test oracles from formal specifications, and integrates these test oracles with a testing framework to automate test executions. This helps make the approach practical, and a blend of formal verification and testing.

A main advantage of my approach is the improved automation of testing process, i.e., generation of test oracles from formal behavioral interface specifications and test executions. I expect that, due to the automation, writing test code will be easier. Indeed, this has been our experience in the JML project. A significant portion of unit testing is automated with the approach, e.g., testing of JML model types [90] and JML specification of Java system classes. It became possible to perform testing as an integral part of programming with minimal effort and to detect many kinds of errors. Almost half of our test failures were caused by specification errors; this shows that the approach is also useful for debugging specifications. The approach also helps make formal methods more practical and concretely usable in programming. One aspect of this is that test specifications and target programs can reside in the same file. I expect that this will have a positive effect in maintaining consistency between test specifications and the programs to be tested, although this remains to be empirically verified. Another advantage is that the approach can achieve the effect of both black-box and white-box testing. White-box testing can be achieved by specifying in-line assertions, predicates on internal states in addition to pre- and postconditions. Assertion facilities such as the `assert` statement are an example of in-line assertions; they are widely used in programming and debugging. Finally, the approach allows programmers to extend and add their own testing methods to the generated test oracles. This can be done by adding hand-written test methods to testcase classes, subclasses of test oracle classes.

In sum, the main goal of my work in this chapter — to ease the writing of testing code — has been achieved. This chapter also demonstrate that the runtime assertion checker can be used as a basis for developing other JML-based techniques and tools.

Chapter 9

Conclusion

9.1 Future Work

There are several natural extensions to the work presented in this dissertation. They include supporting more features of the JML language, improving performance and usability, establishing theoretical foundations, and applying to other tools.

The translation rules and the JML compiler presented in this dissertation do not yet support such JML features as refinement, model programs, and non-functional properties. From the perspective of specification inheritance, I already discussed several issues and hinted at possible approaches to refinement (see Section 6.8.3). In JML, one can also specify the behavior of a method by writing abstract code, called *model programs*, in a notation similar to the specification statements found in refinement calculus [4]. I believe that a subset of these specification statements can be translated into executable code, but it is not clear at this point how to check the post-state against a method specification written using model programs. Recently, JML introduced several specification constructs to specify non-functional properties like time and space requirements. I believe that some of these non-functional properties can be checked at runtime; a concern here would be to exclude resources consumed by runtime assertion checking itself. Finally, it would be a challenging task to support runtime assertion checking in a concurrent environment, e.g., in a multi-threaded program. The assertion checking code should not interfere with the concurrency control of the program being checked, e.g., when synchronized methods are used in assertions. It is not well understood yet how to specify and check concurrent aspects of programs in JML.

There are two distinct areas of future work with respect to performance: performance of the JML compiler and the compiled bytecode. Unlike Java compilers, the current JML typechecker parses the source files of all recursively referenced types. This affects the performance of the JML compiler, as parsing is one of the most costly tasks in compilation. One solution would be to encode the signature information of JML specifications into bytecode or separate symbol files and to eliminate parsing of recursively referenced types. The JML compiler can also be improved by optimizing compilation passes, in particular, by abandoning the double-round compilation strategy (see Section 1.5.1). Some possible approaches would be to generate parse trees in the typechecked form, to support incremental typechecking, and to generate bytecode instead of parse trees. Eliminating the second round compilation would also bring an added benefit of improving usability; e.g., assertion violation errors would be reported in terms of the original source code by indicating the location of method calls that caused the assertion violations.

The JML compiler uses Java's reflection facility to implement delegation calls (see Section 6.1.3). It would be possible to eliminate from the delegation approach the use of reflective calls, which are costly operations in terms of execution time. One approach would be to borrow a method from the class `Object`, say the method `equals`, to encode dispatching code for classes compiled with the JML compiler. The central idea is to turn the method `equals` into a dispatch method, e.g., if the argument is an instance of a particular class that represents delegation requests; otherwise, the method remains to be an equality comparison method that executes the original code. As a dispatch method, the method `equals` statically invokes the requested assertion method (e.g., the precondition method) defined in that class. With this in place, it would be possible to make delegation calls statically by calling the method `equals` on the object to be delegated, because the method `equals` is defined in all classes. Another approach would be to use Java compilers such as MultiJava [30] that support open classes to add a dispatch method to the class `Object`[1].

In this dissertation, I focused my effort on investigating engineering aspects of runtime assertion checking. I also relied on informal arguments to claim that the JML compiler is faithful to the semantics of JML. However, a formal treatment would be appreciated for the JML compiler to be viewed as providing an operation semantics for JML. The semantic faithfulness could be formulated as the soundness and completeness of translation rules. A set of translation rules is *sound* if it does not produce a false positive [48]; similarly, it is *complete* if it catches all assertion violations. The formal treatment requires the translation rules to be defined formally. I believe that it would be possible to formally prove the soundness of translation for a core subset of the JML language or some interesting aspect like specification inheritance.

Finally, it would be possible to extend the capability of JML-based unit testing. One such extension would be to automatically generate some of test cases that the user has to supply to the generated test classes. This would completely automates unit testing. An intriguing possibility is to randomly generate test cases using the type signature information [35] and filter them statically at compile-time or dynamically at runtime guided by such specifications as method preconditions. Another possibility would be to apply some of the specification-based techniques in this areas (e.g., [16]), utilizing JML-specific features such as `example` clauses that specify formalized examples; an example can be thought of as specifying both a test input and a description of the resulting post-state.

9.2 Summary

The work reported in this dissertation was motivated by the lack of practical use of formal behavioral interface specification languages (BISLs) such as JML. One can use BISLs to write detailed design documents of program modules, and such specifications allow one to clarify and critically evaluate the roles of program modules. In addition, I strongly believe that BISLs can be used in daily programming tasks, such as debugging and testing. Debugging and testing consume a significant fraction of program development and maintenance cost, and inadequate debugging and testing also contribute to quality problems. I have presented an approach that can bring “programming benefits” to BISLs; my approach allows one to use BISLs as practical and effective tools for debugging, testing, and design by contract.

In this dissertation, I have presented detailed approaches for translating JML specifications into runtime assertion checking code. These approaches support many recent advances of BISLs (see below), and are implemented as a JML compiler that compiles Java programs annotated with JML specifications into Java bytecode. The compiled bytecode contains instructions that checks JML specifications, such as preconditions, normal and exceptional postconditions, invariants, and history constraints, and yet it can be treated in the same way as the output of Java compilers. The execution of assertion checking instructions is transparent in that, unless an assertion is violated, and except for performance measures (time and space), the behavior of original program is unchanged. The JML compiler represents a significant advance over the state of the art in runtime assertion checking as represented by design by contract tools such as Eiffel [109] [110] or by Java tools such as iContract [81] or Jass [5], because JML provides a rich set of advanced specification facilities

[1] I thank Curtis Clifton for suggesting this possibility.

such as specification-only declarations [95], specifications of interfaces, stateful interfaces, multiple inheritance of specifications, behavioral subtyping [40] [101], and several forms of quantifiers and the set comprehension notation. The JML compiler clarifies many semantic questions about JML and also provides an operational semantics. JML compiler supports *separate* and *modular compilation*; e.g., a subtype may be compiled separately from its supertypes, and yet the compiled bytecode works regardless of the supertypes' inclusion of assertion checking instructions. In addition, it is not necessary to recompile subtypes when the supertype's specifications are modified.

In Chapter 3, I defined a set of translation rules from JML expressions into assertion checking code. The translation rules can handle various kinds of undefinedness such as runtime exceptions and non-executable constructs. The idea is to find as many assertion violations as possible, and the strategy for doing this came from thinking of runtime assertion checking as a game and applying an optimal strategy for selecting a value for undefinedness. For this, undefinedness is classified as either demonic undefinedness or angelic undefinedness; *demonic undefinedness* (e.g., runtime exceptions) is viewed as a potential error whereas *angelic undefinedness* (e.g., informal descriptions) is not viewed as a potential error. The approach is called a *local contextual interpretation.* It is *local* in that an occurrence of undefinedness is interpreted by the smallest boolean expression that encloses the undefinedness. It is *contextual* in that the value of the smallest boolean expression is determined by the expression's operator and relative position to the top-level assertion, such as pre- or postconditions. For demonic undefinedness, the goal is to falsify the top-level assertions under the rules of logic; for angelic undefinedness, the goal is to make them true. The approach is faithful to the semantics of JML by implementing partial functions as under-specified total functions [56] [90]. It is sound and preserves the standard rules of logic.

The translation rules can also handle various forms of quantifiers and JML's set comprehension notation. The approach is at compile-time to estimate a conservative set or interval of values that is sufficient to determine the value of quantification at runtime. For this, several patterns are used to analyze the structures of quantified expressions. If such a set or interval is identified, the quantified expression is translated into code that evaluates the expression iteratively with the quantified variable bound to each element of the set or interval. Otherwise, the quantified expression becomes non-executable, thus contextually interpreted by its parent expression. The approach does not restrict the syntax, and yet evaluates a significant number of quantified expressions.

In JML, interfaces become stateful, as they can have specification-only declarations such as model and ghost fields. As discussed in Chapter 5, the specification state of an interface is represented as a separate *surrogate class.* Thus, an object's specification state is distributed over the object itself and one surrogate object for each interface that the object's class implements. That is, each instance of an implementing class is associated with a unique surrogate object for each interface that the class implements. The surrogate class is also responsible for checking all the specifications of the interface, by hosting all the assertion checking methods of the interface, such as pre- and postcondition methods, invariant methods, and constraint methods. This means that, in addition to the specification state, the responsibility of assertion checking is also distributed over the class being checked, its superclasses, and the surrogate classes of the interfaces that the class implements. The surrogate approach modularizes assertion checking.

In Chapter 6, I introduced a *delegation approach* to inherit specifications from supertypes. A subtype delegates the responsibility of checking inherited specifications to its supertypes. That is, a subtype's assertion methods call the corresponding assertion methods of its supertypes; such a call is *delegated* in that, if the inherited specifications refer to methods declared in the supertypes but overridden in the subtype, then the subtype's methods are invoked. The delegation approach is *modular* in that interpretations are inherited instead of specification text. It is faithful to the semantics of JML, as the inherited specifications are resolved in the supertypes' environment; e.g., fields are statically resolved and instance method calls are dynamically dispatched. The delegation approach supports multiple inheritance, as a subtype can make delegation calls to more than one supertype; to inherit specifications from superinterfaces, delegation calls are made to the surrogate objects of the superinterfaces, uniquely associated with each object of implementing classes.

In JML, specifications are inherited in such a way as to achieve *behavioral subtyping* [40]. This is reflected in runtime assertion checking by the way that the results of delegation calls are combined by the subtype's assertion methods. JML also distinguishes between *strong* [101] and *weak behavioral subtyping* [39]; in weak behavioral subtyping, a subtype's additional methods are relieved from satisfying inherited history constraints. To support this distinction, special constraint methods, called *strong* and *weak subtyping constraint methods* are introduced; they are to be called by strong and weak subtypes respectively.

In Chapter 7, I opened a new possibility in runtime assertion checking by supporting abstract specifications written in terms of specification-only declarations such model fields, ghost fields, and model methods. The idea is to use several kinds of access methods for specification-only declarations; an *access method* is a helper method used by assertion checking code to access a specification-only field or method. Each reference to the declared member is translated into a call to the corresponding access method. For example, an access method for a model field calculates (or retrieves) an abstract value from the program state; i.e., it is an abstraction function for the model field [63] [100, pp. 70–71]. For a ghost field, a pair of getter and setter methods is used to read and write the ghost field. For a model method or constructor, an access method forwards all client calls to the private method or constructor that is generated for the model method or constructor; for a model constructor, the access method becomes a static factory method that calls the corresponding private constructor. An important contribution here is that one can now write abstract specifications that are easy to understand and amenable to formal verification and reasoning, and yet checked at runtime [90].

I demonstrated the practicality and effectiveness of JML's runtime assertion checking by applying it to automating unit testing [29]. The idea is to view interface specifications as test oracles [2] [123] and to use the runtime assertion checker as the decision procedure of the oracles. The tool based on this idea frees the programmer from writing most unit test code, and significantly automates unit testing of Java programs. It allows one to perform unit testing with minimal coding effort and detects many kinds of errors. The successful implementation of the tool also provides a partial proof that the runtime assertion checker can be used as an effective framework for developing other specification-based tools.

I believe that the techniques and approaches developed in this dissertation are applicable to other object-oriented programming languages such as Smalltalk [54] [83], C++ [46] [145], and C# [159], and to formal BISLs with features like those in JML. The contextual interpretation of undefinedness can be implemented in languages with exception handling mechanisms. The delegation approach can be elegantly implemented in dynamically typed languages like Smalltalk. In a programming language like C++, inheritance of specifications may be tailored to use the inheritance mechanism of the language, as the language allows multiple inheritance of code. The surrogate approach can be used for `abstract` classes in C++. Finally, the access method approach to specification-only declarations can be adapted to various languages once the delegation approach works.

The runtime assertion checker, implemented by the JML compiler, brings the benefits of BISLs into programming by helping in localizing errors during debugging and automatically generating test oracles for unit testing. Through automatic checking of consistency between specifications and code, the runtime assertion checker also helps programmers maintain their specifications so that they are up-to-date with respect to the code. Thus, the runtime assertion checker has the potential for decreasing the cost of debugging and testing.

The JML compiler and the JML/JUnit tool described in this dissertation are available from the JML home page `http://www.jmlspecs.org`.

Bibliography

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

[2] Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[3] Karine Arnout and Raphael Simon. The .NET contract wizard: Adding design by contract to languages other than Eiffel. In *Proceedings of TOOLS 39, 29 July -3 August 2001, Santa Barbara, California*, pages 14–23. IEEE Computer Society, 2001.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[5] 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).

[6] Baumeister. Relations as abstract datatypes: An institution to specify relations between algebras. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 756–771. Springer-Verlag, May 1995.

[7] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[8] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[9] J.A. Bergstra, I. Bethke, and P. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied NonClassical Logics*, 5(2):199–217, February 1995.

[10] J.A. Bergstra and A. Ponse. Process algebra with five-valued conditions. In C.S. Calude and M.J. Dinneen, editors, *Combinatorics, Complexity, and Logic, Proceedings of DMTCS'99 and CATS'99.* Springer-Verlag, Singapore, 1999.

[11] Jan A. Bergstra and Alban Ponse. Kleene's three-valued logic and process algebra. *Information Processing Letters*, 67(2):95–103, 1998.

[12] Gilles Bernot, Marie Claude Claudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.

[13] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, May 2000.

[14] A. Blikle. Three-valued predicates for software specification and validation. *Fundamenta Informaticae*, XIV:387–410, 1991.

[15] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[16] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM, July 2002.

[17] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in smalltalk-80. *SIGPLAN Notices*, 24(10):419– 431, 1989.

[18] Luca Cardelli. Structural subtyping and the notion of power type. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 70–79. ACM, January 1988.

[19] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[20] Manuela Carrillo-Castellon, Jesus Garcia-Molina, Ernesto Pimentel, and Israel Repiso. Design by contract in Smalltalk. *Journal of Object-Oriented Programming*, 9(7):23–28, November/December 1996.

[21] Martin D. Carroll. Metaprogramming in C++. *Journal of Programming Languages*, 4(1):1–20, March 1996.

[22] Nestor Catano and Marieke Huisman. Chase: A static checker for JML's assignable clause. In *Proceedings of Verification, Model Checking and Abstract Interpretation (VMCAI '03)*, LNCS. Springer-Verlag, 2003. To appear.

[23] R.G.G. Cattell. *The Object Database Standard: ODMG–93.* Morgan Kaufmann Publishers, San Mateo, California, 1994.

[24] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *Proceedings of ISSTA 96, San Diego, CA*, pages 62–70. IEEE Computer Society, 1996.

[25] Yoonsik Cheon. Inheritance in Larch interface specification languages, its semantic foundation and formal semantics. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Proceedings of International Refinement Workshop & Formal Methods Pacific (IRW/FMP) '98, 19 September - 2 October 1998, Canberra, Australia*, pages 81–99. Springer-Verlag, 1998.

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

[27] Yoonsik Cheon and Gary T. Leavens. A quick overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.

[28] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong 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.

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

[30] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, October 2000. ACM.

[31] Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs.* Springer-Verlag, New York, NY, 1990.

[32] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. *ACM SIGPLAN Notices*, 24(10):433–443, October 1989. OOPSLA '89 Conference Proceedings, Norman Meyerowitz (editor), October 1989, New Orleans, Louisiana.

[33] Dan Criagen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[34] J. L. Crowley, J. F. Leathrum, and K. A. Liburdy. Issues in the full scale use of formal methods for automated testing. *ACM SIGSOFT Software Engineering Notes*, 21(3):71–78, May 1996.

[35] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. College of Computing, Georgia Institute of Technology, Atlanta, GA, November 2002.

[36] Igor D. D. Curcio. ASAP — a simple assertion preprocessor. *ACM SIGPLAN Notices*, 33(12):44–51, December 1998.

[37] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec 1998.

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

[39] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[40] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

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

[42] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115, Portland, OR, August 1992. USENIX Assoc. Berkeley, CA, USA.

[43] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. *ACM SIGPLAN Notices*, 27(10):16–24, October 1992. *OOPSLA '92 Proceedings*, Andreas Paepcke (editor).

[44] Andrew Duncan and Urs Holzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA, December 1998.

[45] Stephen Edwards. Representation inheritance: A safe form of "white box" code inheritance. *IEEE Transactions on Software Engineering*, 23(2):83–92, February 1997.

[46] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Co., Reading, Mass., 1990.

[47] Robert Bruce Findler and Matthias Felleisen. Behavioral interface contracts for Java. Technical Report CS TR00-366, Department of Computer Science, Rice University, Houston, TX, August 2000.

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

[49] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), September 10-14, 2001, Vienna, Austria*, September 2001.

[50] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Cindy Norris and Jr. James B. Fenwick, editors, *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.

[51] Brian Foote and Ralph E. Johnson. Reflective facilities in smalltalk-80. *SIGPLAN Notices*, 24(10):327–336, 1989.

[52] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

[53] A. Q. Gates, S. Roach, O. Mondragon, and N. Delgado. DynaMICs: Comprehensive support for run-time monitoring. 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.

[54] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley Publishing Co., Reading, Mass., 1983.

[55] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2000.

[56] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan 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.

[57] Pedro Guerreiro. Another mediocre assertion mechanism for C++. In *Proceedings of TOOLS 33, June 5-8, 2000, Munt-Saint-Michel, France*, pages 226–237. IEEE Computer Society, 2000.

[58] Pedro Guerreiro. Simple support for design by contract in C++. In *Proceedings of TOOLS 39, 29 July -3 August 2001, Santa Barbara, California*, pages 24–34. IEEE Computer Society, 2001.

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

[60] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[61] Teruo Higashino and Gregor v. Bochmann. Automatic analysis and test case derivation for a restricted class of LOTOS expressions with data parameters. *IEEE Transactions on Software Engineering*, 20(1):29–42, January 1994.

[62] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

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

[64] Ryszard Janicki and Ridha Khedri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 39(2–3):189–213, 2001.

[65] Ryszard Janicki, David L. Parnas, and Jeffery Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, pages 184–196. Springer-Verlag, 1997.

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

[67] Cliff B. Jones. Partial functions and logics: A warning. *Information Processing Letters*, 54(2):65–67, 1995.

[68] Cliff B. Jones and Kees Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[69] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, New York, NY, October 1991.

[70] JUnit.org. Junit. Available from `http://www.junit.org` (Date retrieved: April 2, 2003).

[71] Murat Karaorman, Urs Holzle, and John Bruno. jContractor: A reflective Java library to support design by contract. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference on Reflection '99, Saint-Malo, France, July 19–21, 1999, Proceedings*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer-Verlag, July 1999.

[72] Miguel Katrib and Jesús Coira. Improving eiffel assertions using quantified iterators. *Journal of Object-Oriented Programming*, 10(7):35–43, November 1997.

[73] Miguel Katrib and Ismael Martinez. Collections and iterators in eiffel. *Journal of Object-Oriented Programming*, 6(7):45–51, November/December 1993.

[74] S. Kent and I. Maung. Quantified Assertions in Eiffel. In *Proceedings of TOOLS PACIFIC 95 (TOOLS 18)*, pages 349–364. Prentice Hall, November 1995.

[75] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[76] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proceedings of OOPSLA '02 Conference on Object-Oriented Programming, Languages, Systems, and Applications*, volume 37(11) of *SIGPLAN Notices*, pages 231–245, New York, NY, November 2002. ACM.

[77] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[78] Donald E. Knuth. Literate programming. *Computer Journal*, 27(2):97–111, May 1984.

[79] B. Konikowska, A. Tarlecki, and A. Blikle. A three-valued logic for software specification and validation. *Fundamenta Informaticae*, XIV:411–453, 1991.

[80] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. *ACM SIGSOFT Software Engineering Notes*, 23(2):143–152, March 1998. ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis.

[81] Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[82] Martin Lackner, Andreas Krall, and Franz Puntigam. Supporting design by contract in Java. *Journal of Object Technology*, 1(3), August 2002. Special Issue: TOOLS USA 2002 Proceedings, pages 57-76.

[83] Wilf R. LaLonde. *Discovering Smalltalk*. The Benjamin/Cummings Publishing Company, Inc, Redwood City, California, 94065, 1994.

[84] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[85] Gary T. Leavens. Inheritance of interface specifications (extended abstract). In *Proceedings of the Workshop on Interface Definition Languages*, volume 29(8) of *ACM SIGPLAN Notices*, pages 129–138, August 1994.

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

[87] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim 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.

[88] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[89] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003.

[90] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, March 2003. To appear in the proceedings of FMCO 2002.

[91] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, October 1990.

[92] 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.

[93] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 520–534. Springer-Verlag, New York, NY, 1997.

[94] Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. *Formal Aspects of Computing*, 10:59–75, 1998.

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

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

[97] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of *SIGPLAN*, pages 246–257, New York, June 17–19 2002. ACM Press.

[98] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–223, November 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[99] Hung-Yu Lin. Executing quantified expressions in the JML run-time assertion checker. Master's thesis, The Pennsylvania State University, Capital College, PA, August 2001.

[100] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[101] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[102] Barbara Liskov and Jeannette M. Wing. A new definition of the subtype relation. In Oscar M. Nierstrasz, editor, *ECOOP '93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer-Verlag, New York, NY, July 1993.

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

[104] Mike A. Martin. Effective use of assertions in C++. *ACM SIGPLAN Notices*, 31(11):28–32, November 1996.

[105] Hidehiko Masuhara and Akinori Yonezawa. An object-oriented concurrent reflective language ABCL/R3: Its meta-level design and efficient implementation techniques. In Jean-Paul Bahsoun, Takanobu Baba, Jean-Pierre Briot, and Akinori Yonezawa, editors, *Object-Oriented Parallel and Distributed Programming*, pages 151–165. HERMES Science Publications, Paris, France, 2000.

[106] James C. McKim and David A. Mondu. Class interface design: Designing for correctness. *Journal of Systems Software*, 23(2):85–94, November 1993.

[107] Kristof Mertens, Nele Smeets, Marko van Dooren, Jan Fockx, and Eric Steegmans. A new semantics for JML signals clauses. Technical Report CW 343, Katholieke Universiteit Leuven, Celestijnenlaan 200A - B-3001 Heverlee, Belgium, June 2002.

[108] Bertrand Meyer. Applying “design by contract”. *Computer*, 25(10):40–51, October 1992.

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

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

[111] Richard Mitchell and Jim McKim. *Design by Contract by Example.* Addison-Wesley, Indianapolis, IN, 2002.

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

[113] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002. The author’s Ph.D. Thesis.

[114] Peter Müller and Arnd Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.

[115] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. Technical Report 02-02a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, October 2002. To appear in Concurrency, Computation Practice and Experience.

[116] Jan Newmarch. Adding contracts Java. In *Proceedings of TOOLS 27, September 22-25, 1988, Beijing, China*, pages 2–7. IEEE Computer Society, 1988.

[117] Isabel Nunes. Design by contract using meta-assertions. *Journal of Object Technology*, 1(3), August 2002. Special Issue: TOOLS USA 2002 Proceedings, pages 37-56.

[118] William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, October 1994.

[119] D.J. Panzl. Automatic software test driver. *IEEE Computer*, pages 44–50, April 1978.

[120] Parasoft Corporation. Using design by contract$^{\mathrm{TM}}$ to automate Java$^{\mathrm{TM}}$ software and component testing. Available from `http://www.parasoft.com/jsp/products/tech_papers.jsp?product=Jcontract` (Date retrieved: April 2, 2003).

[121] D. L. Parnas. Predicate logic for software engineering. *IEEE Transactions on Software Engineering*, 19(9):856–862, September 1993.

[122] Lawrence C. Paulson. *Logic and Computations: Interactive Proof with Cambridge LCF*. Cambridge Tracts in Theoretical Computer Science, Volume 2. Cambridge University Press, 1987.

[123] Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, Mar 1998.

[124] Reinhold Plosch. Design by contract for Python. In *Asia Pacific Software Engineering Conference (APSEC 97) and International Computer Science Conference, December 2-5, 1997, Hongkong*, pages 213–219. IEEE Computer Society, 1997.

[125] Reinhold Plosch. Tool support for design by contract. In *Proceedings of TOOLS 26, 1998, Santa Barbara, USA*, pages 282–294. IEEE Computer Society, 1998.

[126] Reinhold Plosch. Evaluation of assertion support for the java programming language. *Journal of Object Technology*, 1(3), August 2002. Special Issue: TOOLS USA 2002 Proceedings, pages 5-17.

[127] Reinhold Plosch and Josef Pichler. Contracts: From analysis to C++ implementation. In *Proceedings of TOOLS 30*, pages 248–257. IEEE Computer Society, 1999.

[128] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[129] Sara Porat and Paul Fertig. Class assertions in C++. *Journal of Object-Oriented Programming*, 8(2):30–37, May 1995.

[130] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, August 2001.

[131] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[132] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of ISSTA 94, Seattle, Washington, August, 1994*, pages 138–152. IEEE Computer Society, August 1994.

[133] Steve Roach and Ann Q. Gates. Synthesis of runtime constraint monitoring code. In *Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification in conjunction with the International Conference on Software Engineering (ICSE 2001), Toronto, Canada, May 13, 2001*, 2001.

[134] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[135] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, October 2000.

[136] Sriram Sankar and Roger Hayes. ADL: An interface definition language for specifying and testing software. *ACM SIGPLAN Notices*, 29(8):13–21, August 1994. Proceedings of the Workshop on Interface Definition Language, Jeannette M. Wing (editor), Portland, Oregon.

[137] Birgit Schieder and Manfred Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 5(2):73–81, February 1999.

[138] Oliver Schoett. Behavioural correctness of data representations. *Science of Computer Programming*, 14(1):43–57, June 1990.

[139] Murali Sitaraman and Bruce W. Weide. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–22, Oct 1994.

[140] Murali Sitaraman, Bruce W. Weide, and William F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, March 1997.

[141] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, NY, second edition, 1992.

[142] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 3, pages 31–48. Addison-Wesley Publishing Co., Reading, Mass., 1989.

[143] John D. Daniels Steve Cook. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. The Object-Oriented Series. Prentice Hall, 1994.

[144] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[145] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[146] Sun Microsystems, Inc. Java 2 platform, standard edition, v 1.4.1 API specification. Available from `http://java.sun.com/j2se/1.4.1/docs/api/` (Date retrieved: April 2, 2003).

[147] 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).

[148] Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Technical Report 619, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., June 1994.

[149] Yang Meng Tan. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.

[150] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.

[151] Mark Utting and Ken Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June/July*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, New York, NY, 1992.

[152] Tim Wahls, Albert L. Baker, and Gary T. Leavens. The direct execution of SPECS-C++: A model-based specification language for C++ classes. Technical Report 94-02b, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, November 1994.

[153] Tim Wahls and Gary T. Leavens. Formal semantics of an algorithm for translating model-based specifications to concurrent constraint programs. In *Proceedings of the 16th ACM Symposium on Applied Computing, Las Vegas, Nevada*, pages 567–575. ACM, March 2001.

[154] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engineering*, 7(4):315 – 343, December 2000.

[155] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture, Analysis and Design of Reliable Systems.* The Object-Oriented Series. Prentice Hall, 1995.

[156] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley Longman, Reading, Mass., 1999.

[157] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.

[158] David Welch and Scott Strong. An exception-based assertion mechanism for C++. *Journal of Object-Oriented Programming*, 11(4):50–60, July 1998.

[159] Scott Wiltamuth and Anders Hejlsberg. C# language specification. From `http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp` (Date retrieved: April 2, 2003), December 2002.

[160] Jurgen F.H. Winkler and Stefan Kauer. Proving assertions is also useful. *ACM SIGPLAN Notices*, 32(3):38–41, March 1997.