ENHANCEMENTS TO JML AND ITS EXTENDED STATIC CHECKING TECHNOLOGY

PERRY ROLAND JAMES

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

JULY 2009

Your file Votre référence
ISBN: 978-0-494-63424-0
Our file Notre référence
ISBN: 978-0-494-63424-0

Abstract

Enhancements to JML and Its Extended Static Checking Technology

Perry Roland James, Ph.D.

Concordia University, 2009

Formal methods are useful for developing high-quality software, but to make use of them, easy-to-use tools must be available. This thesis presents our work on the Java Modeling Language (JML) and its static verification tools.

A main contribution is Offline User-Assisted Extended Static Checking (OUA-ESC), which is positioned between the traditional, fully automatic ESC and interactive Full Static Program Verification (FSPV). With OUA-ESC, automated theorem provers are used to discharge as many Verification Conditions (VCs) as possible, then users are allowed to provide Isabelle/HOL proofs for the sub-VCs that cannot be discharged automatically. Thus, users are able to take advantage of the full power of Isabelle/HOL to manually prove the system correct, if they so choose. Exploring unproven sub-VCs with Isabelle's ProofGeneral has also proven very useful for debugging code and their specifications.

We also present syntax and semantics for monotonic non-null references, a common category that has not been previously identified. This monotonic non-null modifier allows some fields previously declared as nullable to be treated like local variables for nullity flow analysis.

To support this work, we developed JML4, an Eclipse-based Integration Verification Environment (IVE) for the Java Modeling Language. JML4 provides integration of JML into all of the phases of the Eclipse JDT's Java compiler, makes use of external API specifications, and provides native error reporting. The verification techniques initially supported include a Non-Null Type System (NNTS), Runtime Assertion Checking (RAC), and Extended Static Checking (ESC); and verification tools to be developed by other researchers can be incorporated. JML4 was adopted by the JML4 community as the platform for their combined research efforts.

ESC4, JML4's ESC component, provides other novel features not found before in ESC tools. Multiple provers are used automatically, which provides a greater coverage of language constructs that can be verified. Multi-threaded generation and distributed discharging of VCs, as well as a proof-status caching strategy, greatly speed up this CPU-intensive verification technique. VC caches are known to be fragile, and we developed a simple way to remove some of that fragility.

These features combine to form the first IVE for JML, which will hopefully bring the improved quality promised by formal methods to Java developers.

Acknowledgments

This thesis would not have been possible
without the love and encouragement of my family.

I would also like to thank

Dr. Patrice Chalin for the sure guidance, professionalism, interest, dedication, and friendship demonstrated during our years working together.

George for all the fruitful discussions and for his friendship during our studies.

Stuart for his help with TAing and for setting up and maintaining the server hardware.

Leveda for her help with the implementation of the distributed provers.

Dan, Steve, Rajiv, Kianoush, Fred, and Asif (as well as all the above) for their camaraderie as part of the DSRG.

The Faculty of Engineering and Computer Science for awarding me with their Graduate Scholarship.

The Graduate School for granting me the Doctoral Thesis Completion Award.

The Québec Fonds de Recherche sur la Nature et les Technologies for the funding provided.

Contents

List of Figures

List of Tables

List of Acronyms

ADT	Abstract Data Type
AST	Abstract Syntax Tree
ATP	Automated Theorem Prover
BISL	Behavior Interface Specification Language
CbC	Correctness by Construction
CFG	Control-Flow Graph
CIL	Common Intermediate Language
CPU	Central Processing Unit
DSA	Dynamic Single Assignment
ESC	Extended Static Checking
FSPV	Full Static Program Verification
GC	Guarded Command
HOL	Higher-Order Logic
IDE	Integrated Development Environment
ITP	Interactive Theorem Prover
IVE	Integrated Verification Environment
JDT	Java Development Toolkit
JML	Java Modeling Language
KLOC	Kilo Lines of Code
NNTS	Non-Null Type System
OUA-ESC	Offline User-Assisted ESC
OO	Object Oriented
RAC	Runtime Assertion Checking
SOFL	Structured Object-Oriented Formal Language
VC	Verification Condition
VCSE	Verification-Centric Software Engineering
VDM	Vienna Development Method
VSI	Verified Software Initiative
WP	Weakest Precondition

Chapter 1

Introduction

Despite our increasing dependence on software, its general quality remains low. The defects that remain in delivered software cause a range of problems, from simple daily annoyances to the more significant losses of money or even lives. The U.S. Department of Commerce estimated that the impact to the American economy of faulty software in 2002 was $60 billion [Hoare, 2003b]. Since Intel's loss of nearly half a billion dollars because of a faulty division algorithm in a Pentium processor in 1994, formal methods have been successfully used to a much larger extent in hardware design [Schumann, 2001]. It is generally believed that the increased use of formal methods in software development would also result in increased quality [Liu, 2004].

The formal software-development techniques referred to in this thesis are the use of *formal specifications* to express the meaning of a piece of software and the *formal verification* that a given piece of code correctly implements its specification. Specification languages differ from implementation languages in that they are more abstract and allow for the description of what is to be computed without regard for how the computation is to be carried out. Formal verification makes use of theorem provers, either fully automated or interactive [Liu, 2004].

Even though the use of formal methods in software development is seen as potentially beneficial, the associated costs are generally perceived as too high for use in non-safety-critical applications [Schumann, 2001]. Among the reasons for the prohibitive cost of using these quality-enhancing techniques are the lack of tool support and the specialized training required. These issues are being addressed by the research community on two fronts: by providing appropriate tools that ease the burden on software developers and by developing methodologies that allow for the incremental adoption of formal techniques in software written using mainstream languages.

Rushby gives four levels of rigor in the application of formal methods [1993].

1. Without formal methods, specifications are written in prose or pseudo-code, and any analysis is either informal or by means of testing.

2. A slight improvement over this includes the use of mathematical notation in comments (or other documentation) for more succinct and precise expression.

3. By using a formal specification language, automated tool support, such as for type checking, becomes possible.

4. At the most rigorous level, a formal specification language that both has a full semantics and is amenable to formal proof is used in a "comprehensive support environment."

Extending this notion one step further leads to a Verifying Compiler, as described in Hoare's proposed Grand Challenge 6 [Hoare, 2003b]. A Verifying Compiler is one that can "check the correctness of the programs that it compiles" [Hoare, 2003b]. Just as automated checking of syntax and types has eliminated entire categories of runtime exceptions, extending this checking to include behavioral

aspects of programs should greatly reduce the number of semantic errors in code. Much of the necessary theory for program verification has been developed over the past half century. Theorem provers are now sophisticated enough—and computers are now fast enough—to make the Verifying Compiler possible [Hoare, 2003b]. Both the Verifying Compiler and the Grand Challenge 6 projects have been subsumed into the Verified Software Initiative (VSI)[1] [vsi, 2008].

Some existing tools can be seen as early Verifying Compiler prototypes including those supporting

- SPARK [Barnes, 2006],
- Omnibus [Wilson *et al.*, 2005],
- Spec# [Barnett *et al.*, 2005], and
- JML [Leavens *et al.*, 1998]

The first two work with a severely restricted subset of an existing language so that complete specifications can be given and verified using existing techniques, while the last two are ongoing research projects whose eventual goal is the ability to fully specify programs written in a mainstream language.

Promoting incremental adoption of formal techniques precludes forcing developers to change their programming language, so the Java Modeling Language (JML) was created to bring formal techniques to Java developers. JML's syntax and semantics are similar to those of Java, allowing the user to specify both the interface and behavior of Java code [Leavens and Cheon, 2005]. Current tool support for JML includes Runtime Assertion Checking (RAC), Extended Static Checking (ESC), Full Static Program Verification (FSPV), as well as unit-testing tools and documentation generators for specification browsing [Burdy *et al.*, 2005b].

[1] A full list of acronyms can be found on page xvii.

JML has been the focus of many research groups for some years, but there is still much to be done to make its use mainstream. The JML language is still evolving, and there are open issues related both to its soundness and its ability to specify all constructs found in the full Java language. A more pressing problem is that the current generation of JML tools are separate, non-interacting, command-line programs. Providing easy access to these tools through a modern Integrated Development Environment (IDE) should encourage their wider adoption. The research addressed by this thesis aims to enhance both the JML language and its tool support to help ease the adoption of formal methods by mainstream Java developers. As such, this work contributes to Hoare's Verifying Compiler Grand Challenge.

1.1 Problem statement

JML is the de facto specification language for Java [Kiniry *et al.*, 2006]. Its initially stated goals included for it to be "capable of being given a rigorous, formal semantics, and [it] must also be amenable to tool support" [Leavens *et al.*, 2000].

Despite this, two main obstacles remain that prevent JML from being used by mainstream developers:

- the JML language has well recognized deficiencies [Chalin *et al.*, 2006], and
- the original JML command-line driven tool set is showing its age—not yet fully supporting Java 5 despite its having been released several years ago.

Deficiencies in the design of JML itself, prevent developers from writing comprehensive specifications for real-world programs. Like anyone else, developers are more likely to use easy-to-access tools with which they can interact in familiar ways.

1.2 Contributions

> Πολλῶν ἀχύρων ὀλίγον καρπὸν ἀνήγαγον
>
> (*From much chaff I have taken up only a little harvest*)
>
> — Greek proverb

It has been our intent to remove some of the barriers to the widespread adoption of JML by developers by

- [**tool**] developing an Eclipse-based framework within which an enhanced JML tool set could be built,
- [**technique**] developing improved verification tools incorporating the latest theoretical advances and verification techniques as well as novelties developed in the context of this thesis, and
- [**language**] enhancing the JML language rendering it more expressive or complete.

In particular, we make the following novel contributions:

OUA-ESC [**technique**] A main contribution of this thesis is the introduction of a new form of static verification called Offline User-Assisted ESC (OUA-ESC), which is positioned between the traditional, fully automatic ESC and interactive FSPV. With OUA-ESC, automated theorem provers are used to discharge as many VCs as possible, then users are allowed to provide Isabelle/HOL proofs for the sub-VCs that cannot be discharged automatically (Section 5.2).

- Thus, users are able to take advantage of the full power of Isabelle/HOL to manually prove the system correct, if they so choose.

- Exploring unproven sub-VCs with Isabelle's ProofGeneral has proven very useful for debugging code and their specifications.

Monotonic non-null [**language**] Monotonic non-null fields are a common category that has not been previously identified. Once they are initialized, they remain non null, but unlike non-null fields, they are not necessarily initialized during construction. (Section 7.2). We proposed syntax and semantics of this reference modifier for inclusion in JML. Support for it was implemented as part of the Non-Null Type System that we added to Java's compiler, including both static analysis and runtime checks.

Faster ESC [**technique**] We introduce two main ways of making the CPU-intensive verification technique of ESC faster:

- Multi-threaded generation and distributed discharging of sub-VCs are used to speed up the ESC processing (Chapter 6).
- Proof-status caching is also used to reduce the number of calls to theorem provers, thus reducing the time needed for already-verified code (Section 4.2.3)

While neither of these techniques is novel, their application in the context of ESC is.

Reducing fragility of VC caching [**technique**] Proof-status caches are normally susceptible to small changes in their corresponding source code, since the stored VCs contain source-code position information. We propose some simple ways to eliminate this (Section 4.2.3.2).

Enhanced ESC [**technique**] When ESC4 is unable to discharge the VC for an entire method, it breaks that VC into smaller pieces (sub-VCs) and sends these

to several provers (Section 4.2). (Appendix A shows an Isabelle/HOL proof that this decomposition is sound and complete.) One of the consequences of this splitting is that better error reporting is achieved: We are often able to report the subexpression that causes an assertion to fail instead of indicating the entire asserted expression (Section 4.2.1). We also improve usability by indicating provably false assertions.

JML4 [tool] To be able to carry out the work underlying the previously identified contributions, we needed to develop a suitable platform. The result was JML4, which is an Eclipse-based Integrated Verification Environment for JML (Chapter 3). It is the only proposal evaluated by the JML community that satisfies all of its goals for a next-generation of tools. Developers expect an Integrated Development Environment (IDE) to provide (at a minimum) a syntax-sensitive editor, compiler, debugger, and documentation processor as well as support for unit testing and a help facility. Providing access to JML's tools through Eclipse makes its adoption more likely. This framework provides a common base on which the diverse teams working on JML tools can build.

Once the foundational support for JML was implemented atop Eclipse, we began working on the next generation of verification tools. The first provided a rudimentary Runtime Assertion Checking by emitting executable bytecode to check some of the JML annotations. Then a separate compiler phase was added just before code generation that allows for static verification. Others in our research group are developing a Full Static Program Verification component [Chalin *et al.*, 2008a, Karabotsos *et al.*, 2008], but our work on static verification has focused on ESC4, an the Extended Static Checker component of JML4.

ESC4 [technique] & [tool] ESC4 is a complete rewrite of (part of) the functionality provided by ESC/Java2 [Cok and Kiniry, 2005]. ESC4 is a quickly evolving research platform that provides support for some constructs not supported in the earlier tool (Section 5.1).

Verified Software Repository Candidate Part of the motivation for developing JML came from our use of the then-current tools to begin preparing ESC/Java2 for inclusion into the Verified Software Repository (Section 3.1 and Appendix D).

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 covers both background material necessary for understanding later chapters and related work done by others. Chapter 3 presents the development of JML4, its motivation, its architecture, and validation of its validity. Chapter 4 presents the architecture of ESC4, principly how VCs are generated and discharged. Chapter 5 presents some of the benefits of using multiple provers, including Offline User-Assisted ESC. Chapter 6 discusses ESC4's multi-threaded generation and distributed discharging of VCs, and initial timing results are presented. Chapter 7 discusses Monotonic non-null references. We conclude in Chapter 8 by summarizing the work covered in this thesis and presenting future work.

Chapter 2

Background and Related Work

In Section 2.1 we provide a background for the research that will be presented in later chapters as well as survey related work that targets Java. JML and related work for languages other than Java are treated in Sections 2.2 and 2.3, respectively. A survey of tools for use with Java other than JML can be found in Appendix B

2.1 Concepts and Related Projects

2.1.1 Verification-Centric Software Engineering

There are two main approaches to Verification-Centric Software Engineering: the use of

- Behavioral Interface Specification Languages (BISLs) or their lightweight subsets and
- Correctness by Construction (CbC).

As mentioned in the introduction, the formal development techniques to be investigated in this thesis deal with the formal specification of the meaning of code and the formal verification that the code correctly implements its specification. Such specifications are written using a BISL, in our case the Java Modeling language (JML).

CbC is an approach to formalization that is more widely used for the development of safety-critical systems and is exemplified by VDM [Bjørner and Jones, 1978, Jones, 1990] and the B Method [Abrial, 1996]. In these methods, a formal model of an entire system (or subsystem) is developed during the requirements-gathering phase. This abstract model then undergoes a series of refinements to eventually reach an implementation in code. Each refinement step requires a proof that properties of interest in both the input and output models are equivalent [Potter *et al.*, 1996]. CbC techniques have a very high up-front cost, since a full formal model must be developed very early and the necessary proofs are often nontrivial. SOFL was developed as an attempt to mitigate some of these concerns [Liu *et al.*, 1998, Liu, 2004], but we feel that it is still too heavyweight to be widely adopted by industry.

Another group of methods for software development that was not studied are those in which source code is automatically generated from specifications. PerfectDeveloper is among these [Carter *et al.*, 2005]. We ignored these methods since we wanted to take advantage of the redundancy in having two complementary statements of the intended behavior of the system. Because of the difference in the two languages (i.e., specification and programming), it is unlikely that a developer would introduce the same error in both. Without two kinds of artifacts, we cannot usefully exploit this redundancy [Clarke and Rosenblum, 2006].

The approach that we have chosen is lightweight compared to CbC. As discussed further in the next chapter, BISLs are an extension of simple assertions and Design by Contract (DbC). Even though JML and other BISLs can be used to fully specify the behavior of code, and therefore can be considered more than lightweight, they can be used to exploit the lightweight approach of their antecedents—while using the same tools. This allows for the gradual introduction of formalization, as permitted by cost and training constraints. Once initial benefits are seen, it is hoped that more complete specification and validation would follow. This is in sharp contrast to CbC approaches, in which most of the cost comes early and there is not an obvious path to ease developers into formalization.

In the remainder of this section we define BISLs in the context of lightweight verification. An in-depth exploration of JML is given in Section 2.2. An overview of other BISLs for Java can be found in Appendix B.

2.1.2 Assertions, Design by Contract, and BISLs

BISLs are an extension of the notation that supports Design by Contract (DbC), which is itself an extension of program assertions.

2.1.2.1 Assertions

Assertion statements are found in many mainstream languages and are commonly used by programmers [Chalin, 2005]. Hoare notes that "An assertion is a Boolean formula written in the text of a program, at a place where its evaluation will always be true—or at least, that is the intention of the programmer" [Hoare, 2003a]. Many languages that support assertions also allow the developer to provide a text message to be output if the assertion fails. In Java these take the form

```
assert p "useful message about p not holding";
```

The use of assertions as a means of checking code against its expected behavior has a long history. One of the first recorded attempts is found in Turing's "Checking a Large Routine" [Turing, 1949]. This short paper provides a set of assertions that should hold at various points during the execution of a factorial function that performs multiplication by repeated addition. The routine is also shown to terminate by indicating a value that must decrease after each iteration and by showing that the iteration stops when this value reaches zero. One of the main differences in the process suggested in this foundational paper from that advocated today is that Turing's assertions were to be checked by hand.

2.1.2.2 Design by Contract (DbC)

Floyd-Hoare Logic provides a means of reasoning about blocks of code [Hoare, 1969]. These are expressed as a triplet with the form

$$\{P\}\, Q\, \{R\}$$

Dijkstra notes that we can think of these three time-related steps "input; manipulation; output" [Dahl *et al.*, 1972]. Hoare gave the reading, "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion" [Hoare, 1969]. When each construct in a language, including composition, is specified in this manner it becomes possible to describe the meaning of the code Q in terms of the environment P in which it was run.

There are two forms of correctness: total and partial. Partial correctness of Q in this case means that if Q terminates then R will hold. Total correctness of Q means that it is both partially correct and terminates [Winskel, 1993]. In the example by Turing mentioned above, total correctness is shown, whereas if the routine had not been shown to terminate then it would only have been a proof of its partial correctness.

Floyd-Hoare Logic was adapted and popularized under the name Design by Contract (DbC) by Meyer in the Eiffel programming language [Meyer, 1997]. DbC is a relatively easy-to-learn style of specification in which methods are annotated with contracts in the form of two kinds of assertions [Meyer, 1995]:

- **preconditions** that must hold before a method is called and
- **postconditions** that are guaranteed to hold after a method terminates.

Before calling a method, a client must first ensure that the precondition holds. In return for this, the method guarantees that the postcondition will hold. Postconditions often need to refer to the value of a variable before the execution of a method and the value it returns, and mechanisms are usually provided to access these (e.g., `\old(x)` and `\result` in JML). Classes can be annotated with **invariants**, which are properties that must be seen to hold at all times by client code. Invariants can be thought of as being checked in conjunction with all pre- and postconditions.

DbC allows for blame to be assigned when a contract is violated (i.e., when an assertion does not hold). If a method's precondition does not hold when it is called, the client code is to blame. If a method is called when its precondition holds and yet it is unable to satisfy its postcondition then the method itself contains an error [Meyer, 1997]. This blame assignment depends on the not-necessarily-valid assumption that the specification is correct.

Inheritance is an important concept in Object-Oriented (OO) programming, and for programs that make use of inheritance to be easily understood, subtypes must be substitutable for their supertypes. This requirement is known as the Liskov substitution principle and is ensured in DbC by limiting the kinds of contracts that can be made for inherited methods: Preconditions are not allowed to be strengthened, and postconditions are not allowed to be weakened [Liskov and

Wing, 1994]. Overriding methods may weaken preconditions (i.e., require less) and to strengthen postconditions (i.e., deliver more). One of the great benefits of behavioral subtyping for static verification is that even programs with complex inheritance hierarchies can be handled modularly. More specifically, verification tools only have to ensure that (i) at a method-call site, the calling method is correct with respect to the statically declared type of the target, and (ii) that all overriding methods honor the overridden method's contract. This separation eliminates the need for the reasoning about a method call to know about all (or any) subclasses of the method target's dynamic type, so client code does not have to be reverified when subtypes are added to a system [Dhara and Leavens, 1996].

2.1.2.3 Behavioral Interface Specification Languages

DbC allows for the specification of much of a method's behavior, but not all. In addition to what is possible with DbC, Behavioral Interface Specification Languages (BISLs) are able to express which fields a method may modify as well as the exceptions it may throw and the conditions under which these may be thrown. While DbC can specify the behavior of individual methods, it lacks the ability to clearly specify the aggregate behavior of a module. CbC methods provide for the specification of modules as a consequence of their specification of the entire system. BISLs fill this abstraction gap between CbC and DbC in that they allow for the specification of methods and modules. Leavens notes that "[a] *behavioral interface specification* describes both the details of a module's interface with clients, and its behavior from the client's point of view" [Leavens *et al.*, 1999]. They are not meant to specify entire systems, but their strength lies in their ability to capture "detailed design decisions or documentation of intended behavior" of modules [Leavens *et al.*, 1999].

2.1.3 Tool support for DbC and BISLs

While there is quite a lot of tool support for development using BISLs in general and JML in particular [Burdy *et al.*, 2005a], the three main classes of tools from a verification point of view are Runtime Assertion Checking (RAC), Full Static Program Verification (FSPV), and Extended Static Checking (ESC).

2.1.3.1 Runtime Assertion Checking (RAC)

> "Beware of bugs in the above code; I have only proved it correct, not tried it." — Donald Knuth

It is often useful to have a dynamic check of the code and its specification, even though a static proof of correctness would be for all possible executions and the dynamic check is only for a given set of inputs. For example, if a static verification tool is unable to show that a routine is correct, the problem could be either a limitation in the tool's deductive abilities or that the specification and code do not match. Checking some specific cases is often enough to either expose an error in the code or its specification or to boost confidence in their correctness so that continuing to search for a proof is worthwhile.

Tools that support Runtime Assertion Checking (RAC) instrument the code with an executable version of its specification. A method's preconditions and its class's invariants are checked before its body is executed, and its postconditions and invariants are checked after the body terminates. Executing the instrumented code allows the detection and pinpointing of the location of contract violations, even if the violation does not cause an error visible to the user. RAC may be used at any stage, from initial development to "debugging, testing, and production use" [Clarke and Rosenblum, 2006].

Unit testing has become an important and common part of software development in recent years, but developing tests is both difficult and time consuming. Merging unit testing with formal techniques introduces the possibility of storing the description of the intended behavior of a piece of code in both its specification and its unit tests. To both eliminate this unnecessary duplication and ease the burden on the test writer, the specifications can be used to create test oracles: Each data type is given an associated set of values, either defaults or as provided by a developer. Permutations of these values are provided as inputs to a method. If a combination of parameters violates the method's preconditions then it is ignored. If the precondition is satisfied then it can be considered a proper test and the postcondition is checked upon the method's termination [Cheon and Leavens, 2001].

2.1.3.2 Full Static Program Verification (FSPV)

> "Program testing can be used to show the presence of bugs, but never to show their absence." — Edsger Dijkstra

Full Program Verification has as its goal to formally prove the correctness of an implementation with respect to its specification. To do this, a compiler-like tool converts the annotated source to proof obligations whose discharge equates to such a proof. FSPV tools may use automated theorem provers to discharge most proof obligations, but in general, interactive theorem provers are still needed to discharge them all [van den Berg and Jacobs, 2001].

Van den Berg and Jacobs give many advantages of automating program verification: "Theorem provers are very precise, they can do lots of, often boring proof steps in a few seconds, they keep track of the list of proof obligations which are still open, and do a lot of bureaucratic administration for the user" [van den Berg

and Jacobs, 2001]. For the proof obligations to be correct, the translation tool must be based on a formal semantics of both the programming and specification languages. Once the tool has been shown to be correct, it can quickly and accurately produce the complex proof obligations [van den Berg and Jacobs, 2001].

2.1.3.3 Extended Static Checking (ESC)

FSPV is theoretically capable of finding all errors in a system, but it is very expensive. To use FSPV, the program must be specified in great detail, the user of the theorem prover must have quite a bit of specialized training to be able to guide the theorem prover, and each set of proofs takes time to develop. This led in the early 1970s to the introduction of a level of verification between FSPV and simple type checking known as Extended Static Checking (ESC) [Detlefs *et al.*, 1998]. ESC is a compiler-like analysis that can find many kinds of common errors that are not detectable by type checkers or tools that do not make use of specifications, such as `lint`.

As stated earlier, the goal of FSPV is to show that a program is completely correct with respect to its specification. The goal of ESC tools is quite different: They are meant to find mistakes in the code, either discrepancies between the code and its specifications or runtime exceptions that may be thrown and not handled [Flanagan *et al.*, 2002].

A developer can quickly and easily find significant errors by using ESC. Flanagan et al. note that the "static detection of many errors ... is undecidable" in the general case, but that experience has shown that enough errors can be caught for this verification approach to have value [Flanagan *et al.*, 2002].

Unsoundness is built into ESC tools, as each is known to not try to detect certain errors. For example, none of those reviewed below try to detect out-of-memory errors. Similarly, time limits are usually placed on how long a theorem prover is allowed to work with each Verification Condition (VC). When the time limit is reached, the search for a counterexample ends, but it is still possible that the VC is invalid [Leino, 2001].

ESC is usually performed by analyzing the code and its specifications, generating VCs from these, and using an automated theorem prover to discharge the VCs by showing that their negation is false. If a VC cannot be discharged then there is a potential problem to report, but, because of the incompleteness of ESC tools, the problem could be the result of an incomplete specification. Likewise, the theorem prover's inability to prove that a VC's negation is false does not necessarily mean that the corresponding code is correct, since automated theorem provers are known to not be able to prove all that is true [Flanagan *et al.*, 2002].

The theorem prover used in ESC must be completely automatic. That is, the Automatic Theorem Prover (ATP) must require no interaction with the user. As we have seen, interactive theorem provers require specialized training to use them. Also, interaction would reduce the likelihood of the tool being used as often. Leino notes that the lack of interaction is not an unacceptable burden on automated theorem provers for ESC since the VCs to be checked are large but not "mathematically deep" [Leino, 2001].

Since ESC works by deduction instead of by monitoring actual runtime behavior, it is necessary to distinguish between two types of assertions:

- **Assumptions**, which can be taken as givens or that the ESC tool is not expected to be able to prove and

- **Assertions**, which the tool must either show to hold or report in warning messages.

ESC/Moula-3, the forerunner of ESC/Java and ESC/Java2, introduced modular checking as a means of limiting the amount of the system that must be analyzed at one time [Flanagan *et al.*, 2002]. This is achieved by processing each method individually. To check a method, its preconditions are first assumed. Then each statement in its body is checked for possible runtime errors, such as null-pointer dereferences and out-of-bounds array indexes. Any method calls within the one being analyzed are replaced with their contracts. The called method's preconditions are asserted (to ensure that the method being analyzed complies with its side of the contract), and its postconditions are assumed (since analyzing that method is outside the analysis's current scope) [Flanagan *et al.*, 2002]. This allows library code whose implementation is unknown—but whose behavior has been specified—to be reasoned about. Moreover, the tool can "check the uses and implementations of a class without needing all its future subclasses" [Leino, 2001] Loops, as expected, cause problems for ESC systems, and there is much work currently underway to address this issue [Flanagan *et al.*, 2002, Barnett and Leino, 2005, Leino and Logozzo, 2005].

2.2 JML: Language and Tool Support

The Java Modeling Language (JML) is a BISL that was designed for use by Java developers having only modest mathematical training [Leavens *et al.*, 1999]. JML's syntax is similar to that of Java, and its annotations are given in specially formatted comments [Leavens and Cheon, 2005]. Leavens, Baker, and Ruby state that JML is "more expressive ... than Eiffel and easier to use than VDM and Larch,"

and it combines the best features of these other approaches. Compared to the Larch BISLs, JML is simpler and easier to understand since its syntax provides a level of familiarity to Java developers. Use of the Larch specification language requires that developers learn a new notation for assertions that is quite different from Java [Leavens *et al.*, 1998].

JML can document both the interface provided by Java code and its behavior, so it is well suited to documenting detailed designs. Interface specifications include annotations on declarations with extended type information, such as that used by universal types or the non-null type system. The behavioral specifications often specify pre- and postconditions that state properties that should hold when the Java code is executed as well as which—and under what conditions—exceptions may be thrown by the code. In keeping with its goal to allow for incremental adoption, JML allows specifications to range from being detailed and complete to being as little as a single clause giving a single property [Leavens *et al.*, 2000].

Several tools have been developed that process JML-annotated code. These tools provide support for Runtime Assertion Checking (RAC), Extended Static Checking (ESC), Full Static Program Verification (FSPV), automatic discovery of invariants, automated unit testing, and documentation generation [Burdy *et al.*, 2005b].

In the following subsections we present the JML notation and provide an overview of existing JML tools for dynamic checking and static verification.

2.2.1 Language

Design by Contract (DbC) is only one type of specification possible with JML, as JML allows specification-only declarations, which give it the full expressiveness

of model-based languages such as Larch BISLs or VDM. JML hides the mathematical objects used in modeling, such as sets and sequences, behind Java interfaces in a standard library that can be imported. Specification developers are thus relieved of the burden of creating or fully understanding the specifications of these library classes. This allows them to access the power of Larch-style specifications without requiring the same mathematical sophistication [Leavens *et al.*, 1998].

JML provides a notation for the detailed design of Java classes and interfaces. JML notation is given in comments that start with an "at sign" (i.e., either `//@` or `/*@ ...@*/`). An advantage of JML over Larch is that Java syntax is used for the assertion clauses instead of another specification language. The expressions are *pure* Java expressions extended with quantifiers and other logical constructs [Leavens *et al.*, 1998]. A *pure* expression is one that has no side effects [Barnett *et al.*, 2004]. Meyer recommends that Eiffel assertion clauses be pure, but the compiler does not enforce this [Meyer, 1997], while the JML type checker enforces purity [Barnett *et al.*, 2004].

Specifications in JML can be at any level of detail that is needed, from a few properties to full specification. To support this, JML has the concept of *lightweight* and *heavyweight* specifications. If certain keywords are used (such as `normal_behavior` or `exceptional_behavior`) then the specification is considered to be *heavyweight.* In the absence of these keywords, the specification is considered to be *lightweight.* If a heavyweight specification is given, it is taken that the entire behavior has been described and meaningful default values are used for any missing clauses. On the other hand, if a lightweight specification is given then missing clauses are usually taken as being not specified.

Very few keywords are needed to write lightweight DbC specifications. Class invariants are introduced with `invariant`. `requires` precedes preconditions and `ensures` precedes postconditions. Any class fields to be modified in a method can be listed in a `modifiable` clause. The `old` notation, borrowed from Eiffel, is used to refer to the pre-state of a variable [Leavens *et al.*, 1998]

JML supports abstract models in the form of specification-only members. In the JML comments, data members marked with the keyword `model` are not implemented, but are treated within the specification as normal fields. The keyword `initially` introduces an assertion about the state of a model field after the class's constructor has been executed. `depends` and `represents` can be used to show the relation of model fields to implementation fields. A `depends` clause indicates that a model field may change when an implementation field changes value, and a `represents` clause indicates how it will change [Leavens *et al.*, 1998]

Redundancy similar to that provided by Larch's code implies sections can be included in a JML specification. `invariant redundantly`, `requires redundantly`, and `ensures redundantly` introduce assertions that the specifier believes should be deducible from other specifications. `for_example` clauses give specific values. These can be given either as checks for the validity of the specification or for documentation purposes [Leavens *et al.*, 1998].

Behavioral subtyping is ensured by specification inheritance: A class inherits its supertypes' invariants as well as the specifications of their non-private methods and fields. An overriding method's preconditions are or-ed with those of its ancestors, and its postconditions are guarded with an implication of its precondition and and-ed with those of its ancestors [Leavens *et al.*, 1998]. The style of behavioral subtyping produced by this form of specification inheritance is known as *weak behavioral subtyping* [Dhara and Leavens, 1996].

Many other features are commonly used, such as quantifiers, summation and count operators, visibility restrictions, data groups, and initial attempts at specifying concurrency [Leavens *et al.*, 1998]. Java's type system has been expanded to support non-null types [Chalin and James, 2007] and a universe type system [Dietl and Müller, 2005]. The first holds the promise of eliminating null-pointer exceptions, while the latter aims to ease the burden of static analysis by adding structure to the runtime heap.

The JML language is capable of specifying much of the behavior of Java code, including memory usage limits and maximum virtual-machine cycles taken by a method. To reduce the demands on tools that support JML, several language levels were introduced. These range from a minimal set of constructs that every tool should support at Level 0 to esoteric and experimental constructs at the highest levels [Leavens *et al.*, 2008]. We aim to concentrate on the features in Level 0, along with an examination of the concepts of *purity* and *immutability*.

2.2.2 JML Checker and Compiler

One of the advantages that comes from having JML as common language for collaboration is the wide variety of tools that have been produced that support it [Burdy *et al.*, 2005b]. As was seen in the previous subsection, when a single person or group develops their own notation, tool support is usually limited to the immediate confines of their research interests. Our research area is verification-centric software development, so our interest lies in the tools that support RAC, ESC, and FSPV.

The first tool developed for JML was `jml`, a syntax and type checker [Burdy *et al.*, 2005b]. Some of the other JML tools do not provide meaningful output if the code being analyzed would not pass this Checker.

The JML Compiler, `jmlc`, instruments JML-annotated code with executable versions of the specifications to provide RAC. Because of the limitations of automated reasoning systems, RAC currently provides a more practical method of program verification than either ESC or FSPV.

When a method is called, its precondition is checked. If this fails, a precondition-violation exception is thrown. (This is known as an *entry* precondition-violation exception, to distinguish it from an *internal* one, which is caused by a precondition failure for a method called from within this method's body.) A method either terminates normally or by throwing an exception. If it terminates normally, the method's post condition is checked. If an exception is thrown, the exceptional post conditions are checked. Any old expressions in the post conditions are evaluated during the checking of the precondition and cached in local variables until needed. Following Eiffel's lead, the RAC does not check the contracts associated with method calls from within an assertion.

When compiling a method `m`, it is renamed, for example, to `orig$m`. A wrapper method with name `m` is generated that performs the runtime checks. When the original method is called by a client, this wrapper method is executed instead. First, the method's precondition and invariant are checked. If either fails then the appropriate precondition violation exception is thrown. If both succeed, the original method `orig$m` is called inside a try block. If it terminates normally then the return value is stored in a variable local to the wrapper method and the post conditions are checked. If the original method throws an exception then the exceptional postcondition is checked and, depending on the outcome, either the original exception is thrown or the original exception is wrapped in a postcondition violation exception, which is then thrown. Finally, the class invariants are

checked again and, if this succeeds, the value returned by `orig$m` is returned by the wrapper.

The RAC supports the execution of quantifiers. For integral types, every member of the type can be tested. For reference types, all the objects of that type that have been instantiated are used [Cheon and Leavens, 2002].

2.2.3 JMLUnit

JMLUnit was developed as a way to exploit both JML annotations and the JUnit unit-testing framework to automate much of the work involved in writing unit tests [Cheon and Leavens, 2001]. After a developer supplies examples of each of the data types used as targets or parameters of the methods in a class, the JUnit framework is used to evaluate tests created from the JML specification.

Two files are generated for each Java type `T`:

- `T_JML_TestData`, which contains data to be used in the test cases. This file must be edited by a developer to provide representative sets of each data type passed as a parameter to `T`'s methods.

- `T_JML_Test`, which contains the test oracle. That is, "It holds the code for running the tests and for deciding test success or failure" [Cheon and Leavens, 2001].

The test-oracle class is implemented as a subclass of the test-data class. The test-oracle class is the test driver. That is, it selects the data to be used and calls the method under test on the selected target with the selected parameters. This method call is wrapped in a `try` / `catch` block. If an entry precondition-violation exception is caught then the data selected does not make a meaningful test. If an internal precondition- or a postcondition-violation exception is thrown then

an error has been detected and the test has failed. If any other type of exception is thrown then the test has passed, since if the exception had not been part of its specification then it would have caused a postcondition-violation [Cheon and Leavens, 2001].

Some of our recent work was to allow the many test-data files to access a single repository for common types (e.g., `int`, `String`) instead of forcing these to be copied into every class that makes use of them. This also simplifies the maintenance of the test-data files since they may need to be regenerated when a class's methods' signatures change to include new types.

2.2.4 ESC/Java and ESC/Java2

Since ESC/Java2 is similar to a compiler in that it transforms the source language into another form (followed by other processing), it is no surprise that ESC/Java's architecture is a pipeline in which each stage can be analyzed independently (see Figure 1). "The front end produces *abstract syntax trees* (ASTs) as well as a *type-specific background predicate* for each class whose routines are to be checked. The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that routines in that class use" [Flanagan *et al.*, 2002].

Once the AST is produced, each method body is translated into a sugared version of Dijkstra's Guarded Command (GC) language [Flanagan *et al.*, 2002, Leino *et al.*, 1999]. To allow for modular checking, method calls are replaced with translations not of the implementations of the called method but of their specifications. This permits client code to not be reverified when method implementations change, as long as their original specifications are maintained and adhered to [Flanagan *et al.*, 2002].

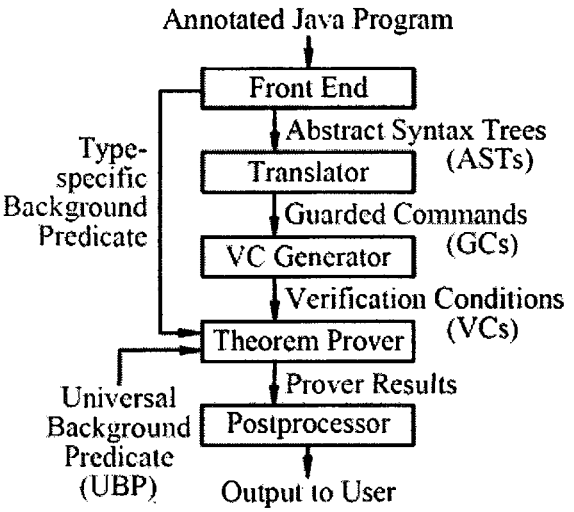

Figure 1: Stages of ESC/Java processing [Flanagan *et al.*, 2002]

The sugared language was found to be useful "in managing complexity and maximizing flexibility" [Leino *et al.*, 1999]. Examples of these benefits include the ability to defer the action of command-line options and the handling of method calls [Flanagan *et al.*, 2002, Leino *et al.*, 1999]. Another difference from Dijkstra's language is that commands may throw exceptions instead of always terminating normally or erroneously [Flanagan *et al.*, 2002].

This first translation to GCs is the source of much of the incompleteness and unsoundness of the tool. For example, integers are modeled as having infinite precision, and loops are not handled rigorously. Since overflow is not considered, later stages of analysis will not know that there are cases in which the sum of two positive integers can result in a value less than zero. Instead of ignoring unspecified loops (i.e., those without a variant and invariant), they are unrolled one-and-a-half times (i.e., the loop is replaced by a check of the condition, the body, and a second check of the condition). "This misses errors that occur only in or after later iterations of a loop" [Flanagan *et al.*, 2002]. Recent changes to

ESC/Java2 cause the user to be alerted when soundness or incompleteness is introduced [Kiniry *et al.*, 2006].

After a method body has been translated to a single primitive GC, a Verification Condition (VC) is produced. A VC "is a predicate in first-order logic" [Flanagan *et al.*, 2002] whose validity ensures that the corresponding code "satisfies its correctness property" [Flanagan and Saxe, 2001]. VCs are generated with an optimized version of the weakest-precondition calculus [Flanagan *et al.*, 2002]. This optimization reduces the size of VCs from being potentially exponential in size of the method to being quadratic worst-case and almost linear in most common cases [Flanagan and Saxe, 2001].

Three pieces are needed by the theorem prover to show the validity of a method R defined in class T:

- a universal background predicate (UBP), which gives properties of Java semantics
- a type-specific background predicate (BP_T), which gives properties of T
- VC_R, which is the VC produced from R

For each method R, the automated theorem prover is tasked with showing that

$$\mathrm{UBP} \wedge \mathrm{BP}_T \Rightarrow \mathrm{VC}_R$$

This formula holds exactly when R is correct with respect to its specification. If the prover is able to show the negation of this formula then a counterexample context is usually output. If the prover takes more than a given amount of time on a method (the default limit is five minutes) then the processing for that method is halted and appropriate output is produced [Flanagan *et al.*, 2002].

A last step involves making the theorem prover's output understandable to the user. VCs that are discharged should cause no message to the user, while those that are not should cause helpful warnings, including the type and location of problem, to be output [Flanagan *et al.*, 2002].

2.2.4.1 Issues with the Provers

ESC/Java2 has until now used Simplify as its automated theorem prover. Simplify was originally developed for use by ESC/Modula 3 and was used in the ESC/Java project [Flanagan *et al.*, 2002].

Simplify is an automated theorem prover with decision procedures that are well suited to deal with equality and arithmetic, and it works by refutation. Validating a conjecture P means searching for variable assignments that allow the negation of P to be true [sim, 2000]. This refutation is the source of the counterexamples mentioned earlier. The theory behind Simplify dates to the 1970s [Leino, 2001], and much work has been done in the area since then.

One of the limitations of Simplify stems from its making a strong distiction between *terms* and *formulas*. Certain expressions are only allowed in formulas and not in terms. We will see examples of these in Section 5.1.

2.2.5 FSPV with JML

Surprisingly many tools have been developed that support FSPV with JML. Jack and Loop, two tools that were developed for proving the correctness of smart card applets, are discussed below. Other tools in this category include Bandera [ban, 2009], Jive [Meyer *et al.*, 2000], KeY [Ahrendt *et al.*, 2005], and Krakatoa [kra, 2009].

2.2.5.1 LOOP

Loop is a tool that converts JML-annotated Java to theories for PVS. It is written in 58,000 lines of OCaml, an OO dialect of ML. After lexing and parsing stages produce an AST, an inheritance-resolving pass "establishes relations between classes by resolving unknown types" [van den Berg and Jacobs, 2001] and marks overridden methods and hidden fields as such. Type checking resolves overloaded operators and methods by giving versions with explicit types. Once the AST has been fully decorated, it is converted to abstract logic syntax (ALS), a form from which it is "easy to generate output for any theorem prover that provides (at least) higher order logic" [van den Berg and Jacobs, 2001]. Pretty printers were originally developed that produce output for PVS and Isabelle, but Isabelle support for JML has since been dropped [Jacobs and Poll, 2003]. To reduce the amount of user interaction required, the proof strategies of PVS have been expanded [van den Berg and Jacobs, 2001].

Loop "uses a shallow embedding of Java in PVS." This has made it possible to prove the soundness (but not the completeness) "of all the programming logic [used]" [Jacobs and Poll, 2003]. It also causes the proof to be carried out at the semantic instead of at the syntactic level. This results in the user having to know how the Java and JML semantics were encoded in PVS in addition to the general expertise of guiding proofs in PVS.

2.2.5.2 Jack

Jack is a set of tools integrated into Eclipse that was developed by Gemplus for FSPV of a subset of Java and JML. The Jack converter translates JML-annotated Java to a form usable by tools that support the B Method, specifically the automated and interactive theorem provers that are included with the Atelier B. Each

Java class is translated into a single B machine in which the class's inheritance hierarchy has been flattened and other classes' members are in-lined. B lemmas are generated whose proof indicates that the class's invariants always hold and its methods' postconditions are guaranteed by their preconditions and bodies [Burdy and Requet, 2002]. The Atelier B automated prover is able to discharge, on average, 80% of proof obligations. While it was a goal for the tool to be sound and complete, shortcomings in the weakest-precondition calculus used make it impossible to prove that the lemmas corresponding to method specifications are "necessary and sufficient to ensure the correctness of the [code]" [Burdy *et al.*, 2003].

Jack differs from other FSPV tools for JML in that it both makes use of an automated theorem prover and provides an easy-to-use interface to an interactive theorem prover. Instead of requiring users to learn the B notation, lemmas and their proof status are converted back to Java for presentation. A goal for the project is to eventually allow fuller access to the interactive theorem prover, but as of [Burdy *et al.*, 2003], only two forms of "indicating a false hypothesis" are supported. Until this goal is realized, users must either manually prove the left-over lemmas in the Atelier B interactive prover or accept that the tool provides only a more rigorous form of ESC.

2.3 VCSE: BISLs and International Research Projects

2.3.1 Other BISLs

In this subsection we look at four non-Java approaches to VCSE. A fifth, SPARK, is covered in Appendix C. It may be of interest to note that the Simplify automated theorem prover mentioned in the discussion of Caduceus/WHY, Omnibus, and

Spec# is the same one used by JML's ESC/Java2, but it has no relation to the Simplifier tool that is discussed with SPARK.

2.3.1.1 Caduceus/WHY

Caduceus [Filliâtre *et al.*, 2008] is a tool for the static verification of C programs annotated with contracts similar to those of JML. Verification in Caduceus is essentially a three-step, manual process: C programs are first translated into the language of the Why system [Filliâtre, 2008], then Why is used to translate VCs into the language of a user-selected prover. The supported provers include Simplify, CVC3, and Isabelle/HOL that are used by ESC4 as well as PVS, Coq, Z3, and others. Finally, the user runs or interacts with the selected prover inorder to discharge the VC proof obligations. The user is left to interpret any prover output, including tracking undischarged VCs back to the source. Such an *offline* approach to verification is like that adopted by the JML4 FSPV Theory Generator [Chalin *et al.*, 2008a, Karabotsos *et al.*, 2008] and contrasts with ESC4's fully automated mode of extended static checking.

2.3.1.2 Eiffel

Eiffel is the language and design methodology that introduced tight integration of DbC with a programming language [Meyer, 1997]. Eiffel was originally defined by Bertrand Meyer in 1986 and made into a ECMA standard (ECMA-367) in June 2005 and an ISO standard (ISO-25436) in June 2006 [Mey, 2007]. The term *Design by Contract* is a trademark of Interactive Software Engineering, a company founded by Meyer [bui, 2008].

Eiffel is a pure object-oriented language (i.e., "everything is an object" in Eiffel) that supports multiple inheritance and generic types. Eiffel is meant to produce simple and clear code. As an example, methods can only have a single entry point and a single exit point [Meyer, 1997].

Eiffel has a non-null type system. In its terminology the *null* value is called `void`, a *nullable* type is called a *detachable* type, and a *non-null* type is called an *attached* type. References are taken to be attached unless explicitly marked as detachable. Classes must define a default constructor that eliminates the default found in other languages of initializing object references to `void`. A unique feature of Eiffel is its Object Test, which combines a Boolean expression and a variable declaration. It has the form `{x: T} exp`, where `x` is a read-only variable being declared of attached type `T` and initialized with the non-`void` value of `exp`. The Object Test returns `true` exactly when `exp` evaluates to non-`void` [Mey, 2005]. Meyers claims that Eiffel is the first language to remove the possibility of throwing a null-pointer exception [Meyer, 2005].

Eiffel only provides RAC, as static verification was seen as unattainable because of the limitations both of theorem provers and of the expressiveness of Eiffel assertions. Assertions are executable Boolean expressions augmented only with the `old` construct. Other constructs that would be needed for full program specification, such as quantifiers, are not supported, as one of the design goals of the language was for it to be simple and easy to use. The recommendation that assertions not have side effects is not enforced by the language [Meyer, 1997].

Behavioral subtyping is enforced by Eiffel's simple rules for contract inheritance. A subclass's invariant is and-ed with those of its superclasses. An overriding method's pre- and postcondition are `or`-ed and `and`-ed, respectively, with those of the overridden method [Meyer, 1997].

2.3.1.3 Omnibus

Wilson *et al.* describe Omnibus as being "designed to be superficially similar to Java but easier to formally reason about" [Wilson *et al.*, 2005]. It compiles to Java bytecode and is aimed at component developers, who can use the code's specification as part of its documentation. One of the main differences between Omnibus and Java is that the former uses value semantics (as in many functional languages) while the latter uses reference semantics (as in most imperative languages). Other simplifications include the lack of static data, exceptions, interface inheritance, arithmetic overflow, and concurrency [Wilson *et al.*, 2005]. Because of these simplifications and other reasons, it is unlikely that Omnibus will be widely adopted by industry.

Because of the restricted nature of the language, Wilson et al. were able to relatively easily provide support for ESC and FSPV. A custom IDE was developed that make easier these and other development activities. Among the results of their research is the discovery of problems that arise when reasoning about a class that uses another class that was verified with another method. Several guidelines are provided to avoid or minimize these problems [Wilson *et al.*, September 2005].

Each of the source files in an Omnibus project has an associated verification policy that gives the level of verification required for it, which can be RAC, ESC, or FSPV. After being parsed and type checked, each file is analyzed by the static verifier. The source is translated into the logics of the theorem provers, as required by the verification policy. Simplify and PVS are the two theorem provers used for ESC and FSPV, respectively. Simplify, being fully automated, provides a complier-like interface. PVS requires the user to interactively (i.e., manually) guide the proofs [Wilson *et al.*, 2006].

2.3.1.4 Spec#

Spec# is Microsoft's extension to C# that aims to increase software quality [Barnett *et al.*, 2005]. It is composed of

- the Spec# programming language, a superset of C# that includes a BISL (and therefore support for DbC),
- the Spec# compiler, which includes an annotated library and is integrated with Microsoft's Visual Studio, and
- the Boogie static program verifier, which performs ESC.

C# is a language developed by Microsoft that has many similarities to Java. A key developer of Spec#, K. Rustan M. Leino, is one of the original developers of ESC/Java, so it is not surprising that the lessons learned from that experience were put to use in developing Spec#. Similar to JML, method contracts may contain pre- and postconditions, indications of the exceptions that may be thrown, and data members that may be assigned to. Spec# has a non-null type system, but arrays cannot be declared to have elements of non-null types.

Behavioral subtyping is enforced in Spec#, but its specification inheritance is quite different from that of JML. Preconditions cannot be added, but additional postconditions are allowed. Additional exceptional postconditions can only be given for exceptions in the overridden method's throws set. Frame conditions cannot be changed with a `modifies` clause but can be effectively reduced in a postcondition.

Inheritance of contracts from interfaces is even more different since this introduces the difficulties inherent with multiple inheritance. Both normal and exceptional postconditions are combined in the same way as when they are inherited from classes, "but the inherited specifications must have identical throws

sets" [Barnett *et al.*, 2005]. Frame conditions from an interface must be a superset of that of the class's implementation. Inheritance of multiple preconditions is not allowed.

Unlike in JML, in which invariants are required to hold only on entry to and exit from non-helper methods, invariants in Spec# are required to hold everywhere except within `expose` blocks. For example, the code of `S` in `expose(o) { S; }` may violate `o`'s invariant as long as it is restored before `S` terminates. By default, exposing an object `o` means exposing the invariants of all classes up to `o`'s static type. To expose more, `expose (o upto T)` can be used. An object must be exposed for its fields to be modified. Spec# also supports object ownership (similar to that found in JML's universe type system), and an object `o`'s owner can only change when `o` is exposed.

As mentioned earlier, Spec# has been integrated with Visual Studio, Microsoft's IDE. Its tool support includes a compiler that provides RAC capabilities and Boogie, which provides ESC capabilities. The Spec# compiler's integration with Visual Studio enables the editors to provide syntax highlighting and code completion. In addition, ESC is called when a source file is edited, allowing for statically detected contract violations to be underlined in red, the same way that syntax errors are. The compiler's output is Microsoft's Common Intermediate Language (CIL) bytecode, with the specifications stored as metadata in a language-independent format.

To support RAC, "Spec# preconditions and postconditions are turned into inlined code" [Barnett *et al.*, 2005]. This results not only in more efficient code than that produced by JML's `jmlc`, but it also eliminates the need for the additional methods introduced by `jmlc`. A method that checks the invariant is added

to each class. Extra fields (e.g., that indicate ownership and to which level the invariant holds) "are added to the super-most class that uses Spec# features within each subtree of the class hierarchy" [Barnett *et al.*, 2005]. Not all specified behavior is checked by the RAC. For example, Spec#'s RAC does not enforce that an object whose field is modified must be exposed. The code from specifications is separated so that tools can distinguish between contract and non-contract code.

Spec#'s ESC tool is Boogie. The first step it performs is translating CIL into BoogiePL, "a simple language with procedures whose implementations are basic blocks consisting mostly of four kinds of statements: assignments, asserts, assumes, and procedure calls" [Barnett *et al.*, 2006]. VCs are generated after several transformations have been applied to the BoogiePL code. These VCs are processed by Simplify, the same automated theorem prover used by ESC/Java2—work was underway some time ago to replace Simplify with Zap [Hunt *et al.*, 2005], a theorem prover developed by Microsoft, but there is no indication that this has yet been done [spe, 2007]. Integration with Visual Studio provides for the outputs of the theorem prover to be presented to the user in terms of the original source code. Thus, the only interaction programmers have with Boogie is by editing source code [Barnett *et al.*, 2006].

The ESC performed by Boogie has quite a few differences from that of JML's ESC/Java2. For example, some loop invariants can be inferred automatically, and loops are handled "in a way that preserves the soundness of the analysis" [Barnett *et al.*, 2005]. Because of the difficulty of reasoning about loops using the weakest-precondition (WP) analysis proposed by Dijkstra [Dijkstra, 1976], ESC/Java unrolls loops a fixed number of times, but this introduces unsoundness [Flanagan *et al.*, 2002]. Barnett and Leino provide a sound method of analyzing loops that

is much simpler than Dijkstra's [Barnett and Leino, 2005]. It does this by cutting loops into acyclic segments and adding appropriate `assume` and `assert` statements before using WP to generate VCs. Essential to their process is the availability of strong-enough loop invariants, thus their motivation for automatically inferring them.

Spec# is still a research prototype, but it appears that Microsoft may have plans for embracing a VCSE methodology. Bill Gates in 2002 called software verification the "Holy Grail of computer science" [Jones *et al.*, 2006]. Spec# has many of the features needed to increase the quality of software developed for the .Net platform. It has been extended to form Sing# by adding the low-level constructs needed to develop the Singularity operating system [Hunt *et al.*, 2005] [Fähndrich *et al.*, 2006]. If operating systems are now seen as beneficial targets of formal development, can office suites and more mundane software be far behind?

Many development systems have been created in recent years that show that VCSE is a possibility. Great benefit can be had from tools and techniques that already exist, but there is much work that can be done to increase the likelihood that these will actually be used by mainstream developers. The next subsection mentions some academic and government-sponsored research initiatives to make that happen.

2.3.2 International Academic and Commercial Research

In addition to the projects mentioned above, two international projects deserve mention: The Mobius project and the Verified Software Initiative.

2.3.2.1 Mobius

The European Union has supported software verification for years. The VerifiCard project, which started in January 2001, was tasked with "[developing] tools for the specification and verification of the Java Card platform and its applications" [Ver, 2007] because of the perceived growth in the markets for Smart Cards. The many deliverables that were produced during the three-year mandate include

- a formal semantics for subsets of Java, including μJava [Nipkow *et al.*, 2000] and Bali [Nipkow *et al.*, 2005],
- a formal specification of the Java Card API, and
- the LOOP tool, the first FSPV for an important subset of Java [van den Berg and Jacobs, 2001]

Part of the current Mobius project builds on the lessons learned with VerifiCard and has as its goal "develop the technology for establishing trust and security for the next generation of global computers" [mob, 2009]. The 16 members of the Mobius Consortium share a budget that is a little less than 7 million Euro [Kiniry, 2007].

2.3.2.2 Verified Software Initiative

Hoare has made a case for the Verifying Compiler to be taken up as a Grand Challenge for Computer Science [Hoare, 2003b]. Some of the criteria for a project to be worthy of being a grand challenge include that it be fundamental, astonishing, revolutionary, understandable, inspiring, useful, feasible, and historical. These are exemplified in previous grand challenges from other fields, which include putting a man on the moon, proving Fermat's last theorem, and mapping the

human genome. Both the Verifying Compiler and the Grand Challenge 6 projects have been subsumed into the Verified Software Initiative (VSI) [vsi, 2008].

Most of the necessary pieces for building a Verifying Compiler already exist. Powerful automated and interactive theorem provers are now able to run on hardware fast enough to make the problem solvable [Hoare, 2003b]. The missing pieces and their integration into an easy-to-use system seem to be within reach.

The International Federation for Information Processing (IFIP) has held two conferences on Verified Software: Theories, Tools, Experiments (VSTTE). Most of these conferences's participants will initially be part of the VSI working group.

"The goal of the [VSI] is to establish software verification as a practical and cost-effective technology for ensuring that software is the most trusted component of any software-based system." [vsi, 2008] We believe that a Verifying Compiler will have as great an impact on development methodologies as did unit-testing frameworks such as jUnit. VCSD has the potential to radically change our expectations of software quality.

Chapter 3

JML4: An Integrated Verification Environment for JML

In this chapter we present JML4, an Eclipse-based Integrated Verification Environment for JML.

To motivate the need for such a system, we give our experience using first-generation tools in concert to provide complementary verification of a non-trivial system in Section 3.1. The purpose of this case study was to show that different verification techniques can be complementary: Each tool and technique has weaknesses that may be compensated for by another.

A big lesson learned from this experience was that while using complementary techniques is useful, it is too difficult with the then-existing tools for ordinary developers to make use of it. This further motivated us to create a simpler way for developers to access JML tools. Section 3.2 introduces our solution: JML4. JML4 is an Eclipse-based IVE framework that provides—among other things—a common frontend for use by other JML tools.

An initial validation of our approach is presented in Appendix E. There we describe the use of JML4's Non-Null Type System to gather usage information

that validated Dr. Chalin's claim that references should have a default nullity of non-null. (This study lead to an examination of the uses of null references in Java, and these results are presented in Section 7.)

After the foundation was in place, work began on developing static verification tools atop JML4. Theses efforts are presented in the next part Chapter 4.

3.1 Motivation for Complementary Verification[1]

> "By three methods we may learn wisdom: First, by reflection, which is noblest ..." — Confucius
>
> "Speed can make up for a lot of things." — Roland Bailey

This section motivates the need for JML4, discussed in the next section, by demonstrating that the verification of non-trivial programs is best achieved when complementary techniques are used together.

Here we present a case study in the use of `jmlc`, the JML Runtime Assertion Checker (RAC) compiler, on ESC/Java2, an extended static checker for Java. The purpose of this case study was to show that different verification techniques can be complementary: Each tool and technique has weaknesses that may be compensated for by another. We believe that overall product quality is maximized by the use of such complementary verification tools. In this study, the use of the JML RAC allowed us to uncover deeper problems with the design of ESC/Java2 than was possible with static analysis alone. Some problems that were found with the RAC are discussed, along with tentative and implemented solutions.

[1] This section is based on [Chalin and James, 2006].

3.1.1 Introduction to Case Study

The two main components of the Verified Software Initiative [(UKCRC), 2006, Woodcock, 2006] are the Verifying Compiler (VC) project and the Verified Software Repository (VSR). A Verifying Compiler is envisioned as a tool to be used by mainstream developers to statically prove that a program is correct. The VSR is meant to hold, among other things, examples of early VC prototypes and "challenge codes," i.e., realistic programs in the form of source code, specifications, and documentation that will be usable as benchmarks for the purpose of exercising proposed VC candidate technologies [Bicarregui *et al.*, 2006].

This section reports on our initial work in preparing the first-generation verification tools of the Java Modeling Language (JML) [Leavens *et al.*, 2008] as potential candidates for inclusion in the VSR. Since the verification tools themselves are written using JML-annotated Java, they can serve as challenge codes as well. More precisely, this section presents a case study on the use of the JML Run-time Assertion Checker (RAC) compiler `jmlc` on ESC/Java2, an extended static checker for Java. ESC/Java2 checks Java source against its specifications, which are expressed using JML.

The main goal of this section is to show that overall product quality is maximized by the use of *complementary* verification tools. For example, routine application of ESC/Java2 to itself has resulted in the elimination of common coding and specification errors. The strength of ESC/Java2 is that it performs fully automated verification and offers a familiar compiler-like interface to developers. As is typical with fully automatic checkers, it has compromised completeness and soundness for efficacy. On the other hand, use of the JML RAC allows specifiers to verify (albeit at runtime) most assertions, and hence can achieve a much higher

degree of completeness. As a consequence, use of the JML RAC has allowed us to uncover more problems with the design of ESC/Java2.

The JML RAC and ESC/Java2 offer a familiar, compiler-like interface and conduct verification fully automatically and tend to be the main verification tools used by JML developers. When both tools are used during development, the following informal process has proven useful: Use is first made of ESC/Java2 to eliminate obvious errors. When a specification becomes too involved, ESC/Java2 will report that it is unable to prove, e.g., that a method body satisfies its contract. In this case, one must resort to using the JML compiler. Even though some assertion expressions are not executable (e.g., some forms of quantified expression), the RAC can generally check more specification statements than ESC/Java2. The caveat, of course, is that the compiled code must be run so that as many input cases as possible are exercised. Another tool, JMLUnit, can be used as a test oracle, automatically creating JUnit test cases from JML specifications. In the next subsection we explain how we made use of the JML RAC to further verify the contracts of the ESC/Java2 application classes.

Unfortunately, the separate command-line tools are too unwieldy for the benefits of complementary verification techniques to be realized by general programmers. This motivated the development of JML4, which we shall see in the next section.

The detail of the case study can be found in Appendix D, but a summary is given in the following section.

3.1.2 Summary

Use of the JML RAC enabled us to uncover significant design flaws and inconsistencies that ESC/Java2 was unable to report (either due to its unsoundness or incompleteness). Trying to detect these through a manual code review would have been very tedious. Having a tool with another verification approach allowed us to rapidly see problems buried in the code. As expected, the problems that the RAC reported were non-trivial: ESC/Java2 had already been run on itself, and the trivial problems this exposed had been dealt with.

The issues raised by the RAC had deep implications (e.g., violations of behavioral subtyping). Careful engineering analysis of both the specification violations and the related source code allowed us to resolve the issues without falling into deeper traps.

This next step in our verification efforts allowed us to uncover design issues in the ESC/Java2 front end and type system whose resolution resulted in a system that has higher consistency between its specification and implementation.

We continued to run RAC-instrumented versions of ESC/Java2 to uncover further bugs, and these are detailed in [Chalin *et al.*, 2008d]. We hope to see ESC4 used to analyze the source for the JML Compiler, as it also has a large JML-annotated code base. Using the tools iteratively to analyze both themselves and each other should allow their quality to be enhanced, hence making them more likely potential candidates for inclusion in the Verified Software Repository.

This case study showed that different verification techniques could be complementary: Each tool and technique has weaknesses that may be compensated for by another. But it also highlighted the difficulty in using the first-generation of tools to acomplish this.

The next section presentes JML4, a framework for an Integrated Verification Environment (IVE) for JML.

3.2 A Framework for a JML IVE[2]

> "It is the framework which changes with each new technology and not just the picture within the frame." — Marshall McLuhan

The previous section demonstrated that verification of sizable programs is best achieved when different technologies are used together. Unfortunately, developers trying to do this with first-generation JML tools must use separate command-line applications and deal with problems like the tools accepting slightly different and incompatible variants of JML. Tool consolidation has become vital. This section presents the architecture and design rationale behind JML4, our proposal for an Eclipse-based Integrated Verification Environment (IVE). As a proof-of-concept, JML4 first enhanced Eclipse with JML's non-null type system and the ability to read JML API library specifications. In addition to the basic features expected from an IDE, JML4 provides RAC and allows easy integration of independently developed static-analysis components, such as ESC4 that is presented in Part 4. A discussion of other consolidation efforts and emerging tools is also given.

3.2.1 Introduction

In the previous section, we confirmed (among other things) how RAC and ESC are most effective when used together, particularly when it comes to the verification of sizable systems. Unfortunately, this is more challenging than it should be; one

[2]This section is based on [Chalin *et al.*, 2007].

of the key reasons being that the first-generation tools accept slightly different and incompatible variants of JML—sadly this is the case for practically all of the current JML tools. The top factors contributing to the current state of affairs are

- partly historical—the tools were developed independently, each having its own parser, type checker, etc. and
- partly due to the rapid pace of evolution of both JML and Java.

Not only does this last point make it difficult for individual research teams to keep apace, it also results in significant and unnecessary duplication of effort.

For some time now, the JML community has recognized that a consolidation effort with respect to its tool base is necessary. In response to this need, three prototypical next-generation tools have taken shape: JML3, JML4, and JML5. This paper is an introduction to the architecture and design rationale behind JML4. After discussing the goals to be achieved by any next-generation JML tool base in Section 3.2.2, we move to the treatment of JML4 in Section 3.2.3. Section 3.2.4 presents a discussion and comparison of JML4 with its siblings JML3 and JML5 as well as other tools like the ESC/Java2 plug-in and the Java Applet Correctness Kit (JACK). In Section 3.3 we present some of the extensions to JML4 and other validation of its architectural approach. A summary is given in Section 3.3.3.

3.2.2 Background and Goals

In this subsection we discuss the main goals to be satisfied by any next-generation tool base for JML. Before doing so, we give a brief summary of JML's first generation of tools and why they are not ideal for continued use.

3.2.2.1 First Generation Tools

The first generation JML tools essentially consist of

- The Common[3] JML tool suite, also known to developers as JML2, which includes the JML RAC compiler and JmlUnit [Burdy *et al.*, 2005b],
- ESC/Java2, an extended static checker [Cok and Kiniry, 2005], and
- LOOP a full static program verifier [van den Berg and Jacobs, 2001].

Of these, JML2 is the original JML tool set. ESC/Java2 and LOOP initially used their own annotation language, though they quickly switched to use JML.

Being independent development efforts, each of the tools mentioned above has its own Java/JML front end including scanner, parser, abstract syntax tree (AST) hierarchy, and static analysis code. This is a considerable amount of duplicate effort and code (on the order of 50–100K source lines of code (SLOC)). This became evident as JML evolved, but the main hurdle which has yet to be fully addressed in the first-generation tools is the advent of Java 5 (especially generics).

Lessons learned from JML2 We would like to benefit as much as possible from the lessons learned from the development of the first generation of tools, especially JML2 which, from the start, has been the reference implementation of JML. JML2 was essentially developed as an extension to the MultiJava (MJ) compiler. By *extension*, we mean that

- for the most part, MJ remains independent of JML
- many JML features are naturally implemented by subclassing MJ features and overriding methods (e.g., abstract syntax tree notes with their associated type checking methods);

[3]Formerly known as the Iowa State University (ISU) JML tool suite.

- in other situations, extension points (calls to methods with empty bodies) were added to MJ classes so that it was possible to override behavior in JML2.

We believe that this approach has allowed JML2 to be successfully maintained as the JML reference implementation since 2002 by an increasing developer pool (there are currently 49 registered developers). Since we wish to follow this approach, we should determine what, if anything, went wrong. We believe that a combination of factors, including the advent of a relatively big step in the evolution of Java (including Java 5 generics) and the difficulty in finding developers to upgrade MJ. Hence our approach in JML4 has been to repeat the successful approach adopted by JML2 but to ensure that we choose to extend a Java compiler that we are confident will be maintained (outside of the JML community).

Evolution of IDEs Another important point to be made about the first generation of JML tools is that they are mainly command-line tools, though some developers were able to make comfortable use of them inside Emacs, which some consider an early IDE.

With a phenomenal increase in the popularity of modern IDEs like Eclipse, it seems clear that to increase the likelihood of getting wide-spread adoption of JML, it will be necessary to have its tools operate well within one or more popular IDEs. In recognition of this, early efforts have successfully provided basic JML tool support through Eclipse plug-ins, which mainly offer access to the command-line capabilities of the JML RAC or ESC/Java2.

3.2.2.2 Goals for Next-Generation Tool Bases

We are targeting mainstream industrial software developers as our key end users. From an end-user point of view, we strive to offer a single Integrated (Development and) Verification Environment (IVE) within which they can use any desired combination of RAC, ESC, and FSPV technology. No single tool currently offers this capability for JML. In addition, user assistance by means of the auto-generation of specifications (or specification fragments) should be possible (e.g., based on approaches currently offered by tools like Daikon [Ernst *et al.*, 2007], Houdini [Flanagan and Leino, 2001], and JmlSpec [Burdy *et al.*, 2005b]).

Since JML is essentially a superset of Java, most JML tools will require, at a minimum, the capabilities of a Java compiler front end. Some tools (e.g., the RAC) would benefit from compiler back-end support as well. One of the important challenges faced by the JML community is keeping up with the accelerated pace of the evolution of Java. As researchers in the field of applied formal methods, we get little or no reward for developing or maintaining basic support for Java. While such support is essential, it is also very labor intensive. Hence, an ideal solution would be to extend a Java compiler, already integrated within a modern IDE, whose maintenance is assured by a developer base outside of the JML research community. If the extension points can be judiciously chosen and kept to a minimum then the extra effort caused by developing on top of a rapidly moving base can be minimized.

In summary, our general goals are to provide

- a base framework for the integrated capabilities of RAC, ESC, and FSPV
- in the context of a modern Java IDE whose maintenance is outside the JML community

- by implementing support for JML as extensions to the base support for Java so as to minimize the integration effort required when new versions of the IDE are released.

A few recent projects have attempted to satisfy these goals. In the next subsection, we describe how we have attempted to satisfy them in our design of JML4. The other projects are discussed in Section 3.2.4.

3.2.3 JML4

JML4 initially enhanced Eclipse 3.3 with

- scanning and parsing of nullity modifiers,
- enforcement of JML's non-null type system, both statically and at runtime, and
- the ability to read and make use of the extensive JML API library specifications.

This subset of features was chosen so as to exercise some of the basic capabilities that any JML extension to Eclipse would need to support. These include

- recognizing and processing JML syntax inside specially marked comments, both in `*.java` files as well as `*.jml` files;
- storing JML-specific nodes in an extended AST hierarchy,
- statically enforcing a modified type system, and
- providing for runtime assertion checking (RAC).

Also, the functionality added by the chosen subset of features is useful in its own right, somewhat independent of other JML features (i.e., the capabilities form a natural extension to the existing embryonic Eclipse support for nullity analysis).

In the remainder of this subsection, we present how we extended Eclipse to support JML, appealing at times to the specific way in which the JML4 features described above have been realized.

3.2.3.1 Architectural Overview

Eclipse Eclipse is a plug-in-based application platform. An Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application-specific plug-ins. Well known bundlings of Eclipse plug-ins include the Eclipse Software Development Kit (SDK) and the Eclipse Rich Client Platform (RCP). Although Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins—called the Eclipse Java Development Tooling (JDT)—offering, among other things, a standard Java compiler and debugger.

The main packages of interest in the JDT are the `ui`, `core`, and `debug`. As can be gathered from the names, the core (non-UI) compiler functionality is defined in the `core` package; UI elements and debugger infrastructure are provided by the components in the `ui` and `debug` packages, respectively. One of the rules of Eclipse development is that public APIs must be maintained *forever*. This API stability helps avoid breaking client code. The following convention was established by Eclipse developers: only classes or interfaces that are not in a package named `internal` can be considered part of the *public* API. Hence, for example, the classes

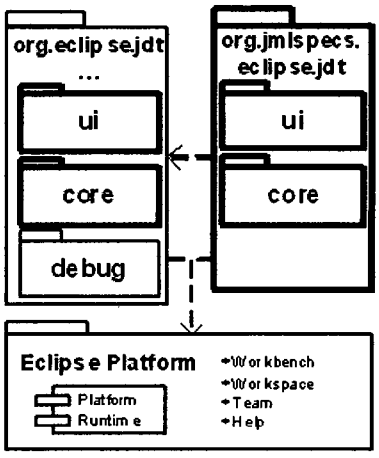

Figure 2: High-level package view

for the JDT's internal AST are found in the `org.eclipse.jdt.internal.compiler.ast` package, whereas the public version of the AST is (partly) reproduced under `org.eclipse.jdt.core.dom`. For JML4, we have generally made changes to internal components (to insert hooks) and then moved most of the JML-specific code to `org.jmlspecs.eclipse.jdt`.

JML4 At the topmost level, JML4 consists of customized versions of the `org.eclipse.jdt.ui` and `org.eclipse.jdt.core` packages (details given below) that are used as drop-in replacements for the official Eclipse JDT `core` and `ui`. These packages are shown in bold in Figure 2.

The complete list of packages that have been customized is given in Figure 3. From this list we can note, among other things, that we have also customized the batch compiler so that JML4 can be used from the command line as well as within the Eclipse GUI. This allows tools based on JML4 to be used both interactively and in batch-processing scripts. Most of the JML-specific modules in the replacement for the `org.jmlspecs.eclipse.jdt` plug-in are subclasses of the Abstract Syntax Tree (AST) node hierarchy, which we examine in greater detail in Section 3.2.3.3, and a few utilities that help with external specifications.

```
org/eclipse/jdt
   /core
     /compiler
   /internal/compiler
     /ast
     /batch
     /codegen
     /flow
     /lookup
     /parser
   /internal/core
     /builder
     /search/indexing
     /util
```

Figure 3: Packages customized to support JML4

3.2.3.2 Overvew of Compilation Phases

The main steps of the compilation process performed by JML4 are illustrated in Figure 4. In the Eclipse JDT (and JML4), there are two types of parsing: in addition to a standard full parse, there is also a diet parse, which only gathers signature information and ignores method bodies. When a set of JML annotated Java files is to be compiled, all are diet parsed to create (diet) ASTs containing initial type information, and the resulting type bindings are stored in the lookup environment (not shown). Then each Compilation Unit (CU) (i.e., source file) is fully parsed to fill in its methods' bodies. During the processing of each CU, types that are referenced but not yet in the lookup environment must have type bindings created for them. This is done by first searching for a binary (`*.class`) file or, if not found, a source (`*.java`) file. Bindings are created directly from a binary file, but a source file must be diet parsed and added to the list to be processed. In both cases, the bindings are added to the lookup environment. If JML specifications for any CU or referenced type are contained in a separate external file (e.g.,

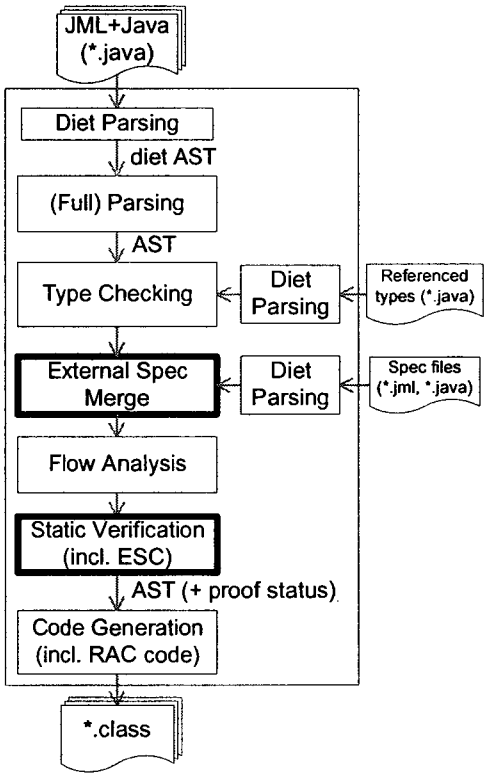

Figure 4: JDT/JML4 compilation phases

a `*.jml` file), then these specification files are diet parsed and the resulting information merged with the CU AST (or associated with the binding in the case of a binary file). Finally, flow analysis and code generation are performed. We anticipate that extended static checking will be treated as a distinct phase immediately following flow analysis.

In the remaining subsections we cover some aspects of JML4 compilation in more detail.

3.2.3.3 Lexical Scanning, Parsing, and the AST

In this subsection we describe some of the particularities of the scanner and parser as well as the approach we have taken to adapting them to support JML.

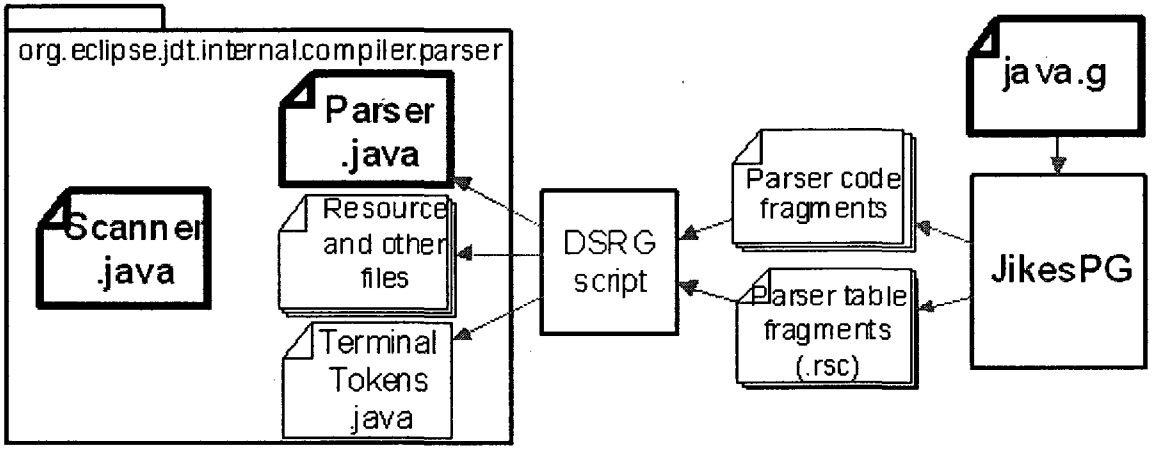

Figure 5: Customizing the JDT lexer and parser

Figure 5 provides an overview of the main parser components as well how they are generated; components in bold are those that have been customized in JML4.

Scanning Since all of JML is contained within specially marked comments, the principle modification to the lexical scanner was to enhance it to recognize JML annotations. This is currently handled using a Boolean field that indicates whether the scanner is in a JML annotation or not. Adding support for new keywords requires a little more work than usual since the JDT's scanner is highly optimized and hand crafted. Keywords, for example, are identified by a set of nested case statements based on the first character of a lexeme and its length. Figure 6 illustrates part of the code added for the recognition of the `non_null` and `nullable` tokens. To make life easier for developers working on JML4, JML's keywords are stored in a `HashMap`, which maps them to the internal token values. To add new JML keyword simply requires adding one line of code that adds an entry to this map.

Parsing The JDT's parser is auto-generated from a grammar file (`java.g`) using the Jikes Parser Generator (JikesPG) (see Figure 5). Instead of producing a

```
...
 switch (data[index]) {
   case 'n' : //non_null nullable \ldots
     switch(length) {
       case 8:
         if (data[++index] == 'o')
             if ((data[++index] == 'n')
              && (data[++index] == '_')
              && (data[++index] == 'n')
              && (data[++index] == 'u')
              && (data[++index] == 'l')
              && (data[++index] == 'l')) {
                  return TokenNamenon_null;
             } else
                  return TokenNameIdentifier;
         else if ((data[index] == 'u')
             && (data[++index] == 'l')
             && (data[++index] == 'l')
             && (data[++index] == 'a')
             && (data[++index] == 'b')
             && (data[++index] == 'l')
             && (data[++index] == 'e')) {
                  return TokenNamenullable;
         }
...
```

Figure 6: Lexer code for nullity keywords

complete parser as a single unit, the JikesPG creates resource files and code fragments that must be incorporated into various other files. We have automated the parser-generation process with a script; the grammar and script reside in a top-level `grammar` directory within the JDT core project, while the `Scanner` and `Parser` are in the compiler's internal `parser` directory. As mentioned earlier, there are two forms of parsing: diet parsing, which only gathers signature information, and full parsing. Both are driven by the same grammar file and use the same Parser class.

```
Nullity → 'nullable'
Nullity → 'non_null'
...
ReferenceType ::= Nullity ClassOrInterfaceType
/.$putCase consumeReferenceType(); $break ./
```

Figure 7: JikesPG grammar productions

Grammar On a positive note, the grammar file, `java.g`, closely follows the Java Language Specification [Gosling *et al.*, 2005]. Somewhat more of a challenge is dealing with the lack of JikesPG documentation. Even though the JikesPG is hosted on SourceForge, the site contains no documentation. To our knowledge the only up-to-date documentation is part of a (master's) thesis written in German [Witte, 2003]. (LPG, apparently a successor to the JikesPG project, is also hosted on SourceForge and has documentation, but it is unclear how different these two applications are. LPG appears to be part of a larger IBM initiative named SAFARI which is described to be "A Meta-Tooling Platform for Creating Language-Specific IDEs". This sounds promising, but SAFARI has yet to be released.)

The `java.g` grammar file has sections defining `Terminals`, `Aliases`, and `Rules`. `Aliases` are used to give readable names to terminals formed from punctuation. An example of production rules for adding support for nullity modifiers to reference types is given in Figure 7. Three rules are shown, and both "->" and "::=" can be used to separate a non-terminal from its definition. Semantic actions come after the production rule between '`/.`' and '`./`' markers. `$putCase` and `$break` are macros that output `case` and `break` statements for the production. Most—if not all—of the semantic actions in the JDT grammar are single calls to methods that are defined in the `Parser` class. This JDT convention helps to both reduce the size of the grammar file and increase its readability.

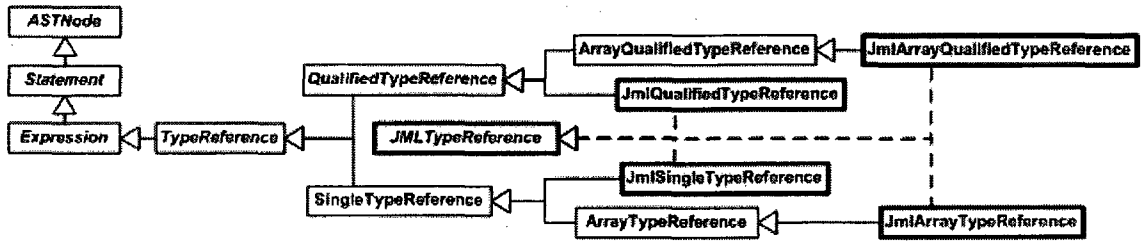

Figure 8: Part of the AST hierarchy(`org.eclipse.jdt.internal.compiler.ast`)

Parser Token Stacks Unlike other auto-generated parsers, those generated by JikesPG do not provide a token stack, and this must be handled in the hand-written code. The `consumeToken()` method is called as each token is processed, and it is used to store information about the current token onto various stacks. Witte provides a summary of the eight stacks maintained by the JDT's parser [Witte, 2003, §3.4.3.3].

Other than modifying the methods corresponding to grammar-rule reductions, the most prominent change to the parser is the replacement of calls to constructors JDT AST nodes by those of JML-specific AST subclasses.

Abstract Syntax Tree Hierarchy The Abstract Syntax Tree (AST) hierarchy for JML4 is obtained by subclassing JDT AST nodes as needed. An illustration of how this is done is given in Figure 8. For example, JML type references are like Java type references except that they have additional information such as nullity. As can be seen, since Java does not allow multiple inheritance, adaptations of the AST can be a bit more involved than merely subclassing.

3.2.3.4 Type Checking and Flow Analysis

Type resolution and checking is performed by invoking the `resolve()` method on a compilation unit. Similarly, flow analysis is performed by the `analyseCode()` method. Addition of JML functionality is achieved by inserting "hooks" into the

previously mentioned methods (i.e., methods with empty bodies in the parent class that are then overridden in JML-specific AST nodes). Our hope is that such hooks will be ported back into the Eclipse JDT, something the JDT developers have confirmed is feasible provided we can demonstrate that no public APIs are changed and that there is little or no impact on runtime performance.

Merging of External Specifications Between type resolution and flow analysis, the compiler checks for external specification files (e.g., `*.jml` files) corresponding to the file being compiled. If one is found, it is parsed and any annotations are added to the corresponding declarations. Binary types (i.e., those found in `*.class` files) whose specifications are needed are handled differently. For these, the system searches for both a source and external specification file. Also, since binary types do not have declarations, but only bindings, information cannot be easily attached to them. For now, we store the specification information about fields and methods of binary types in a cache that is managed by the `JmlBinaryLookup` class.

Non-null Type System JML4's non-null type system is similar to that presented in [Fähndrich and Leino, 2003]. That is, each Java reference type `T` is replaced with a pair of types, `nullable T` and `non_null T`, with `non_null T` a subtype of `nullable T`. We have initially left out support for generic types and concern for the initialization soundness issues addressed with raw types. Work is underway to address these (see, e.g., [Cok, 2008]), but work has yet to be started to address the handling of arrays (e.g., array elements are currently restricted to having the same nullity as the array reference, and assignment of arrays is invariant with respect to nullity).

It was much easier to add a non-null type system to the JDT than it was to the JML2 Checker. This is partly due to the intra-procedural nullity analysis already implemented in the JDT, but even more so because of the ease of adding nullity information to the AST. The JDT's nullity analysis was only concerned with local variables and method parameters, as the latter are implemented in the AST as a subclass of local variables. In the JML2 Checker, it was necessary to modify every AST node that contained a reference to a type to also store its nullity. In the JDT, each mention of a type in the source code is reflected in the AST with a node that is a subclass of `TypeReference`. These were replaced with instances of subtypes of `JmlTypeReference`, which simply add a `Nullity` field. The nullity of a method's return type is stored in a `MethodDeclaration`, and that of all other typed items is stored in subclasses of `AbstractVariableDeclaration`. Since much of the type-checking and flow-analysis code makes use of *bindings* and not *declarations*, modifications were made so that each `FieldBinding` and `MethodBinding` created for a source file stores a reference to its declaration.

Once nullity information was easily accessible, the intra-procedural nullity checking only needed to be modified slightly to add the checking of fields and return types. All dereferences were already guarded during flow analysis by a call to `Expression`'s `checkNPE` method. This method originally ignored cases in which the reference being checked was not to a local variable (or method parameter), so it was modified to also complain when the reference was neither to a local (or parameter) nor declared to be non-null. This required the addition of an `isDeclaredNonNull` method to `Expression` that was overridden in only a small number of its subclasses. We also had to ensure that nullity declarations and information were in fact used. For example, since originally their nullity was taken to be unknown, the declared nullity of a method's parameters must be explicitly

initialized before doing flow analysis on it. Also, local variables were originally set to definitely null, non-null, or unknown, and this was changed so they are either definitely non-null or potentially null.

Assignment was the only case for which there was no built-in support. There are two forms of assignment: explicit assignment to a variable and parameter binding during a method call. Assignment of a reference is only allowed if the `lhs` is declared to be nullable or the `rhs` is known to be non-null. Checking method parameters only required a straightforward change to the behavior of the class `messageSend`: Before doing the original flow analysis, a check is made to determine if the `Expression` representing an actual parameters can be assigned to the corresponding formal `Argument`. When an assignment would not be allowed, an error is reported.

Problem Reporting Eclipse has a sophisticated error reporting subsystem that supports flexible filtering controlled by compiler preferences that are based on individual compiler options and option groups. Nonetheless, adding support for a new problem (i.e., something that can translate into an error or warning to the user) requires only small additions to the compiler's problem reporter and the addition of some minor glue in the form of `int` and `String` constants.

Any violations of the non-null type system are reported using the JDT's problem-reporting mechanisms. This requires adding a new method to the `ProblemReporter` class for each kind of problem. These pass the necessary information to an inherited `handle` method, which presents the problems in the GUI by, among other things, underlining the offending code in yellow or red and adding an entry in the `Problem` view (see Figure 12). When using the JDT's batch compiler, handling a problem causes information to be sent to the standard output (see the expected output in Figure 10).

```
public static void generateNullityTest(CodeStream codeStream,
                                       String exceptionType, String msg) {
  BranchLabel nonnullLabel = new BranchLabel(codeStream);
  codeStream.dup();
  codeStream.ifnonnull(nonnullLabel);
  codeStream.newClassFromName(exceptionType.toCharArray(), msg);
  codeStream.athrow();
  nonnullLabel.place();
}
```

Figure 9: Code generation example for runtime checking of a cast (to non-null)

3.2.3.5 Instrumentation for Runtime Assertion Checking

Code generation is performed by each `ASTNode`'s `generateCode()` method. Its `CodeStream` parameter provides methods for emitting JVM bytecode and hides some of the bookkeeping details, such as determining the generated code's runtime stack usage. Hence, supporting runtime checking is relatively straightforward.

As an example, consider Figure 9, which shows the code from `JmlCastExpression` for checking a cast to non-null. This method tests the top element on the stack against null and throws an exception if it is. First, a label that will be used as a jump target is created. Then, the value on top of the runtime stack is duplicated so that a copy will remain after the execution of the next instruction, which removes the top element and jumps to the given target if the value popped is not null. A new exception of the given type is constructed, with the given message as its parameter, and placed on the runtime stack. The next statement `throws` the item on the top of the stack. Finally, the label for the conditional jump's target is placed after the exception-throwing code.

3.2.3.6 Static Cecking

Hooks have been provided to invoke static checking between the compiler's calls to `analyseCode()` and `generateCode()`. Currently, support is provided for ESC/-Java2, ESC4, and FSPV-TG [Chalin *et al.*, 2008a]. In addition to providing warnings to the user, this allows static checking subsystems to manipulate the AST before bytecode is generated, possibly removing checks for properties that have been proven to always hold. More details are provided in Chapter 4.

3.2.3.7 Testing Framework

Unit tests of both the compile time and runtime checking have been developed. The compile-time tests have been integrated into the JDT's unit-testing framework, and an example can be seen in Figure 10. To create a new set of tests, a subclass of `AbstractRegressionTest` is created. Compiler options can be set by overriding the `getCompilerOptions()` method. The names of the individual test methods begin with the word `test`, which is usually followed by an (ordinal) test number and a descriptive name. The body of the test is often a single method call, to either `runPositiveTest` or `runNegativeTest`, depending on whether any resulting bytecode is to be executed or not. The source of the code to be compiled is in-lined as a `String`, as is any expected output.

3.2.4 Related Work

In this subsection we provide a brief discussion of the tools which have either positioned themselves as next-generation JML tools or at least could be considered potential candidates.

```
public void test_0004_AssignmentExpression() {
  this.runNegativeTest(
    new String[] {
      "AssignmentExpression.java",
      "/*@ nullable_by_default */\n" +
      "\n" +
      "\n" +
      "\n" +
      "class AssignmentExpression {\n" +
      " /*@ non_null */ String non  = \"hello\"; //$NON-NLS-1$\n" +
      " /*@ nullable */ String able = null;\n" +
      "\n" +
      " void m1(/*@non_null*/ String s) { this.non = s; }\n" +
      " void m2(/*@nullable*/ String s) { this.non = s; } //error\n" +
      " void m3(/*@non_null*/ String s) { this.able = s; }\n" +
      " void m4(/*@nullable*/ String s) { this.able = s; }\n" +
      " void m7(/*@non_null*/ String s) { if (s!=null) this.non = s; }\n" +
      " void m8(/*@nullable*/ String s) { if (s!=null) this.non = s; }\n" +
      " void m9(/*@non_null*/ String s) { if (s!=null) this.able = s; }\n" +
      " void m10(/*@nullable*/ String s) { if (s!=null) this.able = s; }\n" +
      "}\n"
    },
    "----------\n" +
    "1. ERROR in AssignmentExpression.java (at line 10)\n" +
    " void m2(/*@nullable*/ String s) { this.non = s; } //error\n" +
    "                                   ^^^^^^^^^^^^\n" +
    "Possible assignment of null to an L-value declared non_null\n" +
    "----------\n" +
    "2. ERROR in AssignmentExpression.java (at line 13)\n" +
    " void m7(/*@non_null*/ String s) { if (s!=null) this.non = s; }\n" +
    "                                           ^\n" +
    " The variable s cannot be null; it was either set to a non-null value " +
    "or assumed to be non-null when last used\n" +
    "----------\n" +
    "3. ERROR in AssignmentExpression.java (at line 15)\n" +
    " void m9(/*@non_null*/ String s) { if (s!=null) this.able = s; }\n" +
    "                                           ^\n" +
    "The variable s cannot be null; it was either set to a non-null value " +
    "or assumed to be non-null when last used\n" +
    "----------\n"
  );
}
```

Figure 10: JML-JDT unit test

3.2.4.1 JML3

The first next-generation Eclipse-based initiative was JML3, created by David Cok. The main objective of the project was to create a proper Eclipse plug-in, independent of the internals of the JDT [Cok, 2007]. In addition, JML3 goals included: providing functionality similar to that made available by command line tools (e.g. checker, RAC, ESC), "classic Eclipse UI enhancements" for JML (e.g. syntax highlighting), as well as support for generation of specifications. Considerable work has been done to develop the necessary infrastructure, but there are growing concerns about the long term costs of this approach.

Due to the closed non-extensible nature of the public JDT extensions points, Cok had to write a separate parser for the entire Java language and AST. As was mentioned earlier, the JDT creates two AST structures, one internal (using nodes from `org.eclipse.jdt.internal.compiler.ast`) and the other part of the public API (`org.eclipse.jdt.core.dom`). The public AST is generated from the internal version, but this conversion is one way. JML annotations are parsed with a custom parser. This JML parser is applied only to the comments found in the source code. The resulting JML AST nodes are used to decorate the original JDT DOM AST, and a second step is needed to match the JML AST nodes to the correct JDT AST nodes.

Cok notes that "JML3 [will need] to have its own name/type/resolver/checker for both JML constructs [and] all of Java..." in addition to the duplicated parser and AST [Cok, 2007]. Since one of the main reasons for integrating JML with Eclipse was to escape from providing support for the base Java language, this is a key disadvantage.

3.2.4.2 JML5

An annotation apparatus was introduced in Java 5 for decorating classes, fields, and methods with meta-data. JML5 is a project, recently initiated at Iowa State University, with the goal to replace JML specifications in Java comments with annotations. Such a change will allow JML's tools to use any Java 5 compliant compiler.

An example of a JML5 specification is shown in Figure 11. It illustrates the use of a JML declaration modifier (`@spec_public`) on the two fields `a` and `b` to make them accessible to specifications in other classes. The two fields are constrained to being positive through the definition of two invariants. These are enclosed within an `@InvariantDefinitions` annotation because a declaration cannot currently have multiple annotations of the same type. Method specifications are enclosed within `@SpecCase(...)` annotations. Here, the method `m` has a normal-behavior heavyweight specification case, as denoted by the `Type.normal_behavior` attribute. `Type.exceptional_behavior` and `Type.behavior`, both with their usual meaning, are also defined in JML5. The absence of a `type` attribute indicates a lightweight specification. Multiple specification cases can be defined with the `@Also` annotation.

Unfortunately, the use of annotations has important drawbacks as well. Java's current annotation facility does not allow for annotations to be placed at all locations in the code at which JML can be placed. JSR-308 (Annotations on Java Types) is addressing this problem as a consequence of its mandate [Ernst and Coward,], but any changes proposed would only be present in Java 7 and would not allow support for earlier versions of Java [Ernst and Coward,]. Additionally, provisions would have to be made to allow for the conversion of the extensive JML libraries to be accessible to the new tools.

```
class Tester {
  private @spec_public int a;
  private @spec_public int b;
  @InvariantDefinitions({
    @Invariant( value = ''a > 0'', msg = ''a is positive''),
    @invariant( value = ''b > 0'', msg = ''b is positive'')
  })

  @SpecCase(type = Type.normal_behavior,
            requires = "n.length == 2",
            ensures = "a == @old(a)+n[0] && b == @old(b)+n[1]")
  public @Pure bool m(int @NonNull [] n) {a+=n[0]; b+=n[1];}
}
```

Figure 11: JML5 example specification

3.2.4.3 Java Applet Correctness Kit (JACK)

The Java Applet Correctness Kit (JACK) is a proprietary tool for JML annotated Java Card programs initially developed at Gemplus (2002) and then taken over by INRIA (2003) [Barthe *et al.*, 2007]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [Burdy *et al.*, 2003].

JACK's main goals are that (1) it should be supported in an environment familiar to developers and (2) it should be easy for Java developers to verify their own code [Burdy *et al.*, 2003]. The first goal is accomplished by providing JACK as an Eclipse plug-in and the second by providing developers with a proof obligation viewer. This viewer is used to communicate the proof obligations along with their associated JML and Java code to the user. To further facilitate ease of use, these proof obligations are displayed in the Java/JML Proof Obligation Language (JPOL). JPOL shares its syntax with Java and JML thus hides theorem prover specific syntax.

The proof obligations are discharged using one of the supported automated and interactive provers, currently the B prover, Coq, PVS, and Simplify. Through the Jack proof viewer a user can see the proof obligation in either JPOL or the prover's native representation. Through this viewer, user interaction is limited to identifying false hypotheses or showing invalid execution paths in the code. If the user is has the required expertise, then the proof obligations native to a specific theorem prover are displayed and the user can interactively attempt to discharge them.

While JACK is emerging as a candidate next-generation tool (offering features unique to JML tools such as byte code verification [Burdy *et al.*, 2007]), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3.

3.2.4.4 ESC/JAVA2 Plug-in

An Eclipse plug-in was developed for ESC/Java2 with the latest release dating to February, 2005 [Cok *et al.*, 2007]. It provides functionality similar to that of the command line tool. Additionally, code and specification elements responsible for verification violations are highlighted and associated with useful error messages in a fashion similar to other Java warnings.

To construct this plug-in, the code base of ESC/Java2 is packaged into a .jar file. Provided a Java file is being edited, options are available to the user to statically verify the code. Upon the user's invocation, the environment is prepared and a top-level method is called that causes the Java source file to be parsed and a verification condition (VC) to be generated and fed to the prover. Violations are then reported in Eclipse.

Simply put, this is a wrapper for the command line tool. Parsing is done using the ESC/Java2 parser. Nevertheless, this is an improvement to the command line

Table 1: A Comparison of possible next-generation JML tools

		JML2	JML3	JML4	JML5	ESC/Java2 Plug-in	JACK
Base Compiler / IDE	Name	MJ	JDT	JDT	any Java 7+	ESC/Java2 and JDT	JDT
	Maintained (supports Java ≥ 5)	×	✓	✓	✓	×[1]	✓
Reuse/extension of base (e.g. parser, AST) vs. copy-and-change		✓	×	✓	×	×	×
Tool Support	RAC	✓	✓	✓	(✓)	N/A	N/A
	ESC	N/A	(✓)	(✓)	N/A	✓	✓
	FSPV	N/A	(✓)	(✓)	N/A	N/A	✓

MJ = MultiJava, JDT = Eclipse Java Develoment Toolkit
N/A = not possible, practical or not a goal, (✓) = planned
[1] ESC/Java2 is currently being maintained to support new verification functionality, but its compiler front end has yet to reach Java 5.

tool simply because it integrates ESC/Java2 with Eclipse and makes it even easier to verify code.

3.2.4.5 Summary

Table 1 presents a summary of the comparison of the tools that have been suggested as possible foundations for the next generation of support for JML. As compared to the approach taken in JML4, the main drawback of the other tools is that they are likely to require more effort to maintain over the long haul as Java continues to evolve due to the looser coupling with their base.

3.3 Early Results and Validation of Architectural Approach

"All the proof of a pudding is in the eating." — William Camden

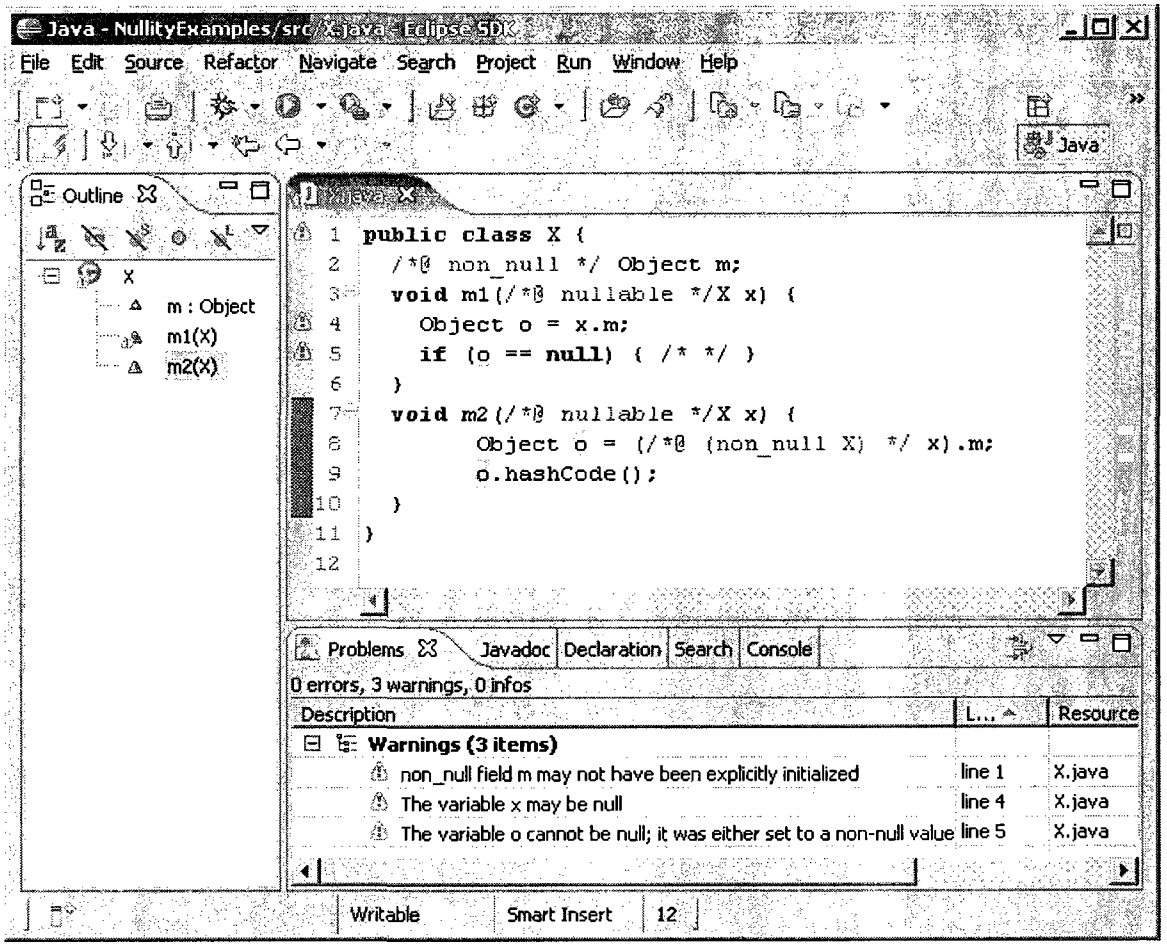

Figure 12: Screenshot of JML4

3.3.1 Use of JML4

JML4 was used to validate our proposal that JML's non-null type system should be non-null by default [Chalin and James, 2007] (summarized in Appendix E). It was used to produce RAC-enabled versions of five case studies (totaling over 470K SLOC), which were then used to execute those systems' extensive test suites. This exercise gave us confidence in the runtime checking and the processing of the JML API specifications. A screenshot of the edit-time and compile-time checking of nullity annotations is illustrated in Figure 12.

Initial support fo Static Checking, including Extended Static Checking and Full Program Verification have been built atop JML4. ESC4 is discussed in Chapter 4, and the Full Static Program Verification–Theory Generator is described, e.g., in [Chalin *et al.*, 2008a] and [Chalin *et al.*, 2008b].

3.3.1.1 Third-Party Features

As mentioned in Section 3.2.2.2, one of the main goals for JML4 is for it to serve as a platform for other research groups. Since success of JML4 will be measured in part by how easily researchers (other than those working on the JML JDT core) can extend JML4. We have already seen some encouraging signs of success, as others have built upon JML4:

RAC Yoonsik Cheon's research group at the University of Texas at El Paso is in the process of building a full scale RAC implementation based on JML4 [Sarcar, 2009].

Symbolic execution and test generation: Serum/Kiasan Robby and his team at Kansas State University are making use of JML4 as a front-end to the Bogor/Kiasan symbolic execution system and the associated KUnit test generation framework [Deng *et al.*, 2007].

Specification execution Tim Wahls is extending JML4 to enable the execution of specifications through the use of constraint programming [Krause and Wahls, 2006, Catano and Wahls, 2009].

Boogie backend for JML4 A group of senior-year Software-Engineering students are developing a static verification system using JML4 as the front end and targeting Boogie as the backend, which they have named JML4/Disco. This should component should allow JML4 to leverage the extensive work done

at Microsoft. At this time, Boogie is only available under MS-Windows, and its source is not available. The first point can be addressed using wine [win, 2009], and there has been some talk about making Boogie an opensource project.

3.3.2 Validation of Architectural Approach

JML4, like JML2, is built as a closely integrated and yet loosely coupled extension to an existing compiler. An additional benefit for JML4 is that the timely compiler base maintenance is assured by the Eclipse Foundation developers. Hence, as compared to JML2, we have traded in committer rights for free maintenance; a choice which we believe will be more advantageous in the long run. Losing committer rights means that we must maintain our own version of the JDT code. Use of the SVN vendor-branch feature has made this manageable.

While we originally had the goal of creating JML4 as a proper Eclipse plug-in, only making use of public JDT APIs (rather than a replacement plug-in for the JDT), it rapidly became clear that this would result in far too much copy-and-change code; so much so that the advantage of coupling to an existing compiler was lost (e.g., due to the need to maintain our own full parser and AST). Nonetheless we were also originally reluctant to build atop internal APIs, which contrary to public APIs, are subject to change—with weekly releases of the JDT code, it seemed like we would be building on quicksand. Anticipating this, we established several conventions that make merging in the frequent JDT changes both easier and less error prone. These include

- avoiding introducing JML features by the copy-and-change of JDT code, instead we make use of subclassing and method extension points;

- bracketing any changes to our copy of the JDT code with special comment markers.

Following these conventions, incorporating the regular JDT updates since the fall of 2006 (to our surprise) has taken less than 10 minutes, on average.

3.3.3 Summary

The idea of providing JML tool support by means of a closely integrated and yet loosely coupled extension to an existing compiler was successfully realized in JML2. This has worked well since 2002, but unfortunately the chosen Java compiler is not being kept up to date with respect to Java in a timely manner. We propose applying the same approach by extending the Eclipse JDT (partly through internal packages). Even though it is more invasive than a proper plug-in solution, using this approach we have demonstrated that it was relatively easy to enhance the type system and provide RAC support.

Other possible next-generation JML tools have been considered [Chalin *et al.*, 2007], but all seem to share the common overhead of maintaining a full Java parser, AST, and type checker separate from the base tools they are built from. This seems like an overhead that will be too costly in the long run. We are certainly not claiming that JML4 is the only viable next-generation candidate but are hopeful that this thesis has demonstrated that it is a likely candidate.

The first JML4 prototype served as a basis for discussion by some members of the JML consortium, and eventually it came to be adopted as the main avenue to pursue in the JML Reloaded effort [Robby *et al.*, 2008]. A JML Winter School followed in February 2008, during which members of the community were given JML4 developer training [Leavens, 2009, Wiki]. Since then, JML4's feature set has

been enhanced, in particular, with support for next-generation ESC (see Chapter 4) and FSPV components.

Even though JML4's approach is currently more invasive than a proper plug-in design, using this approach we have since 2006, been able to (i) maintain JML4 despite the continuous development increments of the Eclipse JDT, and (ii) demonstrate, through the recent addition of the JML SV, that JML4's infrastructure is capable of supporting the full range of verification approaches from RAC to FSPV. Hence, we are hopeful, that JML4 will be a strong candidate to act as a next-generation research platform and industrial-grade verification environment for Java and JML.

In the next chapter we discuss ESC4, JML4's Extended Static Checking component.

Chapter 4

ESC4: A Modern ESC for Java[1]

In the previous chapters, we saw the need for an Interactive Verification Environment (IVE) that provides easy access to various forms of verification. We also saw a realization of this in JML4, an Eclipse-based IVE for JML, and its usefulness was demonstrated.

In this chapter and following two we examine ESC4, JML4's first static verification component. Its goal is to provide a complete rewrite of the functionality of ESC/Java2 while taking into account the advances that have been made in the years since the earlier tool's development. As a starting point, we leverage the JML4 compiler front end and use the abstract syntax tree (AST) that is produced as input to ESC4. This is an immediate improvement over ESC/Java2, which has its own compiler front end that must be maintained. ESC4 does not incur the cost of building the AST, since it is produced as part of the normal compilation process.

This chapter examines ESC4's architecture. Chapter 5 points out some of the design decisions made in the development of ESC4 that allow it to verify code that previous tools cannot. It also introduces Offline User-Assisted ESC (OUA-ESC), a

[1]This chapter is based on [James and Chalin, 2009b].

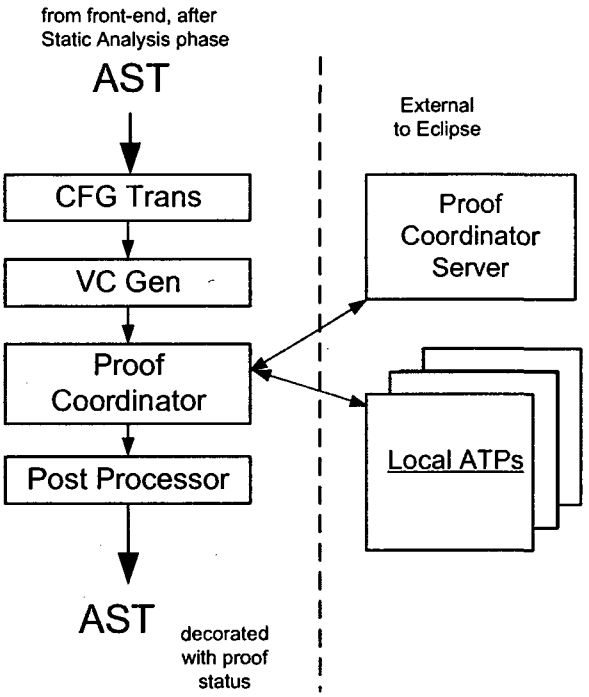

Figure 13: Data flow in ESC4

novel form of static verification. We end the discussion of ESC4 in Chapter 6 with a presentation of a way of speeding up the system: a multi-threaded, distributed version of ESC4.

Figure 13 shows the dataflow in ESC4. The processing begins by translating a method's AST, including its contract and body, first to a passive, acyclic Control-Flow Graph (CFG) and then to a Verification Condition (VC). If a method's VC can be shown to be true then the method body conforms to its contract; otherwise, there may be a violation. ESC4 uses theorem provers to try to automatically discharge VCs. When the theorem provers are unsuccessful, either because the VC is invalid or simply because the theorem provers are not powerful enough to find the proof automatically, contract violations are reported using Eclipse's error

reporting infrastructure. This allows users to navigate to verification failures as easily as they do for syntax errors.

In the following sections, we will look more in-depth at how ESC4 carries out each of the steps in Figure 13. We begin by looking at its architecture. In Section 4.1 we look into the intermediate languages and visitors that are used to produce VCs. In Section 4.2 we look at the various techniques ESC4 uses to discharge those VCs.

4.1 Generating VCs

> "It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."
> and "Make things as simple as possible, but not simpler.")
> — Albert Einstein

4.1.1 Introduction

In this section we look at the architecture of ESC4's VC-generating front end, while the next section presents the VC-discharging back end.

ESC4 is implemented as a compiler stage between JML4's flow analysis and code generation. If the compiler's front end finds any errors (e.g., syntax or typing) in a class then ESC4 does not process it. The ESC4's processing stages are shown in Figure 14. Each method's AST is converted first to a Control-Flow Graph (CFG) as described in [Barnett and Leino, 2005]. This approach allows for the straightforward translation of while loops and other control-flow structures to an

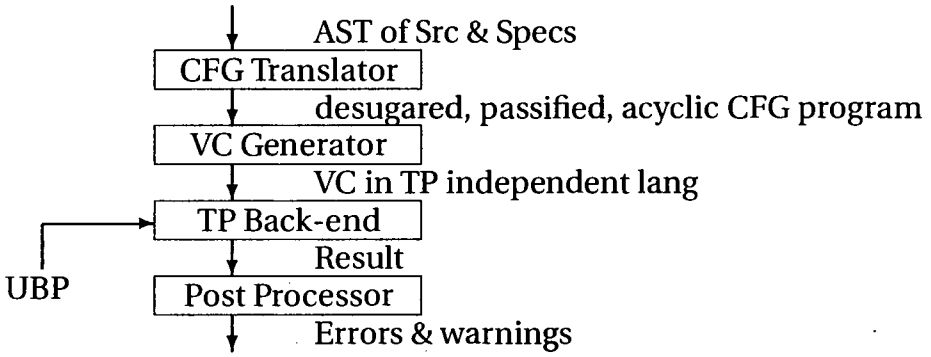

Figure 14: ESC4's processing stages

acyclic control-flow graph. Dynamic Single Assignment is used to remove side effects. Using a weakest-precondition calculus, the passive, acyclic graph that represents an entire method and its specification is converted to a single Verification Condition (VC).

The first stages of ESC4 can be thought of as a compiler, or automated translator, that converts the JML-annotated Java code to VCs. Like many compilers, the architecture of ESC4 is a series of pipes and filters. The various stages transform the source, in our case the JML-decorated JDT AST for a single method, first to a CFG language then into a VC that can be processed by a theorem-prover back end, the `ProverCoordinator`. Finally, the results of the theorem provers are presented to the user in the form of errors and warnings. The first two boxes in Figure 14 are expanded in the following subsections. To illustrate their effects, we use a running example, the method shown in Figure 15[2]. This method is supposed to return the absolute value of the value passed to it. Its contract's lack of a precondition shows that this method should behave properly for any `int` as input. The contract further shows that the value returned will be both non-negative and either equal to the value passed in or its additive inverse. We see that the first

[2]We ignore the special treatment needed to handle the largest negative value to avoid overcomplicating this example.

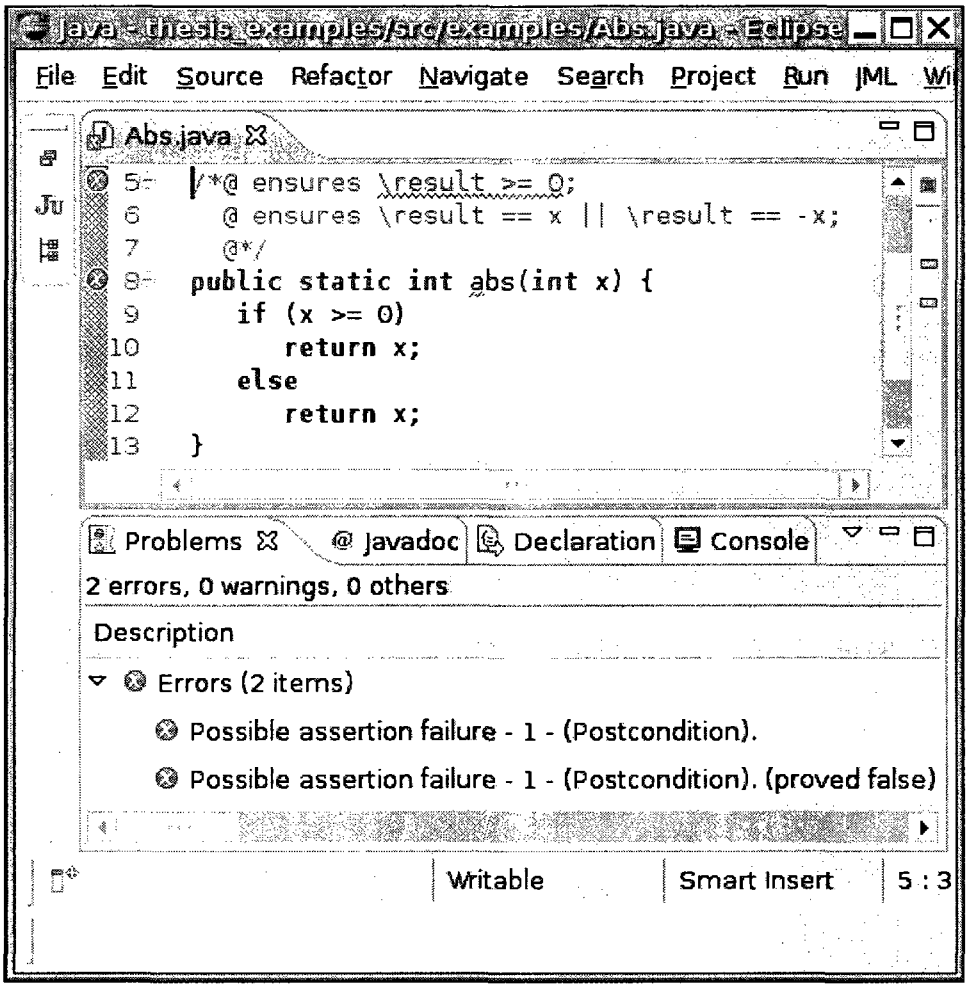

Figure 15: ESC4 reporting a problem with `abs`

postcondition is not respected. Indeed, the second `return` statement is at fault. In the following subsections we will see how ESC4 generates a VC that can be used to determine that the method body does not respect its contract.

4.1.2 Control-Flow Graph Translator

Dijkstra's Guarded Command (GC) language [Dijkstra, 1976] is commonly used as an intermediate language for ESC tools. We instead use for our translation Control-Flow Graphs (CFGs), which are core to the approach presented in [Barnett and Leino, 2005]. Whereas Dijkstra's original formulation works well for

```
if x ≥ 0 ⟶ abs := x
[] x < 0 ⟶ abs := −x
fi;
if   abs > 0 ∧ (abs = x ∨ abs = − x) ⟶ skip
[] ¬ (abs > 0 ∧ (abs = x ∨ abs = − x)) ⟶ abort
fi
```

Figure 16: `abs` in Dijkstra's GC language

structured code, the new approach also works with unstructured code. Normally, Java is seen as a structured language, but some common constructs are inherently unstructured, such as breaking or continuing from the middle of loops, returning from methods at arbitrary points, and handling exceptions. We examine this CFG form before presenting our translation to it.

4.1.2.1 GC Language and Control-Flow Graph

The translation of our example code into Dijkstra's GC language is shown in Figure 16. Note that the square brackets ([]) introduce a nondeterministic choice. In our example, this choice is between two guarded statements, exactly one of whose guard can be true. The guards are the predicates before the arrows (⟶), and the second guard is simply the negation of the first. In addition to `if` and `fi`, the keywords `skip` and `abort` are also introduced. `skip` has no effect and causes processing to continue to the next statement. `abort` causes the evaluation to stop. Statements are separated by the semicolon (;). The first `if` block corresponds to the body, and the second to the postcondition. If no path through the GC code ends in `abort` then the code satisfies its contract.

The approach presented by Barnett and Leino can be used to analyze unstructured code. To do this, they work with a Control-Flow Graph (CFG) in which the nodes are blocks of code that can end with nondeterministic jumps to other

```
start:   x@122$0 :: int;
         return@0$0 :: int;
         assume: (True & True);
         goto: [then$1, else$1]
then$1:  assume: x@122$0 ≥ 0;
         assume: return@0$0 == x@122$0;
         goto: [return]
else$1:  assume: ¬ (x@122$0 ≥ 0);
         assume: return@0$0 == x@122$0;
         goto: [return]
return:  assert(Postcondition): True & ((return@0$0 ≥ 0) &
                                  ((return@0$0 == x@122$0) ? True :
                                                  (return@0$0 == −x@122$0)));
         goto: []
```

Figure 17: `abs` as a Control-Flow Graph

blocks. Instead of guards, `abort`, and `skip`, they use `assume` and `assert` statements. A guard $x \longrightarrow$ is replaced with `assume` x. If an `assume`'s predicate evaluates to false, its execution blocks, but does not otherwise cause problems. If the predicate of an `assert` statement evaluates to false then the whole computation fails. If no path through the CFG leads to an `assert` statement that fails then the code satisfies its contract.

The final form of our example as a CFG program is shown in Figure 17. The first block, named `start`, begins by declaring some local variables of type `int`. The formal parameter `x` is treated as a local variable, as is the value returned by the method, which is stored in the variable named `return`. The first number after the variable names gives the character position for the variable's declaration. It is added to make the variable name unique and is needed when there are multiple variables with the same name. The second number is the incarnation, which will be explained below in Section 4.1.2.5. The implicit precondition and class

invariant are handled by `start`'s `assume` statement. The first statement in the original code is an `if`, which is translated as a nondeterministic jump to the `then` and `else` blocks. Since the names for these blocks are automatically generated, they are suffixed with counters to make them unique. Both these blocks begin with an `assume`: the `then` block assumes the `if`'s original predicate, and the `else` block assumes its negation. The body of each of these blocks is the translation of the original `return` statement and a jump to the `return` block. The `return` block asserts the postcondition, which includes the implicit class invariant.

This translation is not made in a single step, so we now go through the four smaller steps necessary to realize the full transformation. The CFG Translator converts the input AST to a desugared, acyclic, passified CFG program using a sequence of Visitors [Gamma *et al.*, 1995].

- The first step is to remove some unnecessary complexities of the JDT's AST by translating a method to a fully sugared CFG program. (Syntacic sugarings let us delay making some choices until later.)
- Next, statements that affect control flow (e.g., `if`s and loops) are removed to produce an acyclic CFG program.
- A separate step removes any remaining sugarings.
- Finally, a transformation to a Dynamic Single Assignment (DSA) form produces an even simpler CFG program that contain only `assume`s, `assert`s, and `goto`s.

4.1.2.2 From the JDT's AST to ESC4's

The first step towards producing the final CFG program is to translate the JDT's AST to a highly sugared form that removes much of the information not needed

SugaredStmt ::= **assert** *Expr* | **assume** *Expr* | **break** *loopLabel*
| **continue** *loopLabel* | **exprStmt** *Expr* | **goto** *blockId*
| **if** *Expr SugaredStmt SugaredStmt*
| **postcondition** *Expr* | **precondition** *Expr* | **return** [*Expr*]
| **sequence** *SugaredStmt SugaredStmt* | **varDecl** *name type*
| **while** *loopLabel Expr SugaredStmt*

Figure 18: Sugared-Statement Language

for ESC. A grammar for the fully sugared language is shown in Figure 18, where the expressions *Expr* are all those allowed by Java and JML (except for a few, as mentioned below). This translation is done by the `Ast2SugaredVisitor`. This step also removes some of the unnecessary complexity of the JDT's AST. For example, the JDT contains nodes for qualified (i.e., dot separated) names and single names inspired by the source syntax instead of more abstract nodes such as for fields and local variables. This stage replaces the JDT's `Single-` and `QualifiedNameReferences` with `SugaredVariables` and `SugaredFieldReferences`. Also, short-circuiting conjunctions (`&&`) and disjunctions (`||`) are replaced with equivalent conditional expressions (`?:`). All unlabeled loop statements are given an explicit label name, and all default `continue` and `break` statements are made to reference this name. References to the JML keyword `\result` are replaced with references to the new local variable `return`, which is given a declaration source location of 0. More details are provided below.

Loop statements, continues, and breaks can be labeled statements in Java, and we provide default labels for those that were left unlabeled. Removing loops in the next step is then made simpler since all loops and `SugaredBreak` and `SugaredContinue` statements will have labels.

Except as noted above, this step is an almost literal translation. At this stage, control-flow statements such as loops are encoded as simple statements. Formal

```
start:  x@122 :: int;
        return@0 :: int;
        SugaredPrecondition: True & True;
        SugaredIfStatement: x@122 ≥ 0
                                then: [return: x@122]
                                else: [return: x@122]
        goto: [return]
return: SugaredPostcondition: True & ((return ≥ 0) &
                                [(return@0 == x@122) ? True :
                                        (return@0 == −x@122)])
        goto: []
```

Figure 19: Fully sugared version of `abs` as a CFG

parameters are treated the same as other local-variable declarations, except their declarations come at the very beginning of the CFG program and are followed by the method's precondition, which contains the class's invariant. In our example, these two predicates are the implicit `True` (See Figure 19).

Exactly two blocks are produced in this step. The first is given the name `start` and contains declarations of the formal parameters, assumption of the class's invariant and the method's precondition, the translation of the body, and finally a `goto [return]`. The second block is given the name `return` and consists of asserting the class's invariant and method's postcondition.

One kind of statement in the JDT's AST is a `Block`, which contains an array of statements. In the case this array is `null`, it is translated as `assert` True. When there is more than a single element in the array, each of the elements is converted, and the resulting list is folded into a `SugaredSequence` statement.

All local variables and formal parameters are made unique by associating with them the location of their declaration. Java does not allow two local variables

AcyclicStmt ::= **assert** *Expr* | **assume** *Expr* | **exprStmt** *Expr*
| **goto** *blockId* | **havoc** *l-value*
| **sequence** *AcyclicStmt AcyclicStmt*
| **varDecl** *name type*

Figure 20: Acyclic-Statement Language

with the same name to be in scope at the same time, but non-nested scopes can have variables with the same name.

`while` statements can be labeled statements in Java, so we provide a label for unlabeled loops. Similarly, labels are provided for unlabeled `break` and `continue` statements. Removing loops in the next step is made simpler since all loops and `SugaredBreak` and `SugaredContinue` statements have labels. Care must be taken so that breaks and continues from nested loops are labeled correctly.

Conditional-And (`&&`) and Conditional-Or (`||`) Operator expressions are translated to Conditional-Operator (`?:`) expressions. This reduces slightly the number of expressions that must be supported in later stages.

The second step is to convert the fully sugared program to an acyclic Control-Flow Graph with sugared GCs in the blocks that form the nodes.

4.1.2.3 Removing Control-Flow Statements

Once we have a `start` block that contains the fully sugared version of the method body, we remove control-flow statements so that the result is a CFG whose nodes are blocks that end in `goto` statements. In addition, we remove loops in a way that is sound and complete, per [Barnett and Leino, 2005]. A grammar for the acyclic language is shown in Figure 20, where the expressions *Expr* are the same as in the sugared-statement language. Note that there are many fewer constructs in this language, but a **havoc** statement has been added. This step's transformation

is performed by the `DesugarLoopVisitor`. Key to this translation is the use of *loop invariants.* Loop invariants are properties that must hold before a loop is entered and after each iteration of the loop.

Some statements and most expressions are passed through unchanged by this stage. `SugaredPreconditions` and `SugaredPostconditions` are replaced with appropriate assume and assert statements. Sequences of expressions are handled in a special way because most of the constructs that are not passed through will be converted to blocks that end in `goto`s. As is shown below, many new blocks are created for each loop or `if` statement.

A `SugaredIfStatement` is replaced with `goto [then, else]`, and new blocks are created for the two branches as well as for a rendezvous point after the statement. The `then` block is prefaced with an assumption of the condition, and the `else` block with its negation. A missing `else` clause is replaced with an empty statement, which is translated as `assert True`. Both blocks end with `goto [afterIf]`, where `afterIf` contains the statements following the original `if` statement. In our example, the `afterIf` block is not reachable, so it was not shown. As can be imagined, it would only contain the command `goto [return]`.

Loops are a little more complicated to translate than `if`s. Luckily, we are able to handle them more easily than in [Barnett and Leino, 2005] because we start from structured loops. Since the original technique takes as input the equivalent of bytecode, care must be taken to identify all jumps to loop heads. In Java, only the three loop statements can be used to create loops, and only `continue` statements and the end of the loop bodies can cause a jump to the loop head.

During this transformation, a new kind of statement is introduced, the `havoc`, which has the effect of giving its arguments arbitrary values. For example, after

the statement `havoc x`, we know nothing about the value stored in `x` except that it is a value allowed by its declared type.

A `SugaredWhileStatement` is replaced with an assertion of its invariant and a jump to the loop header. The loop header is a point where the invariant must hold, and all jumps to the beginning of the loop are to this header. New blocks are created for the loop header, the loop body, for the code after the loop, and for `break` targets. The loop header `havoc`s the targets of the loop (i.e., variables and fields that are assigned to in the loop condition and body) and then goes nondeterministically to either the `body` or `after` block. The `body` block assumes the loop condition and loop invariants, stores the value of any loop variant expressions, the translation of the loop body, asserts the loop invariant and checks that the variant functions decreases. It ends with a `goto []`, indicating that it is a dead end. It is only evaluated to ensure that the body restores the loop invariant and decreases any loop variants. For any loop variant that is given, we add an implicit invariant that its value is nonnegative. The `after` block assumes the loop invariant and the negation of the loop condition and ends with a `goto [breakTarget]`. `break` statements are translated to `goto [breakTarget]`. `continue` statements are similar to the end of a loop body block and are translated as such. This is shown schematically in Figure 21.

`return` statements with a value are translated to the sequence `return@0 =` *expr*`; goto [return]`, where *expr* is the value returned. Those without a value are simply translated as `goto [return]`.

Each formal parameter v that appears in the postcondition is replaced with its prestate value, which is encoded as `\old(`v`)`.

Method calls are not removed in this phase but during passification (see Section 4.1.2.5).

```
...
codeBefore
maintaining invariant;
loop_variant variant;
while (condition) {
      body;
}
codeAfter
...
```

(a) Original loop

```
                ...
                translation of codeBefore
                assert invariant;
                goto [header]
header:         havoc targets;
                assume invariant;
                goto [body, after]
body:           store variant
                assume condition;
                translation of body
                assert invariant;
                check variants
                goto []
after:          assume ¬ condition;
                goto [breakTarget]
breakTarget:    translation of codeAfter
                ...
```

(b) Acyclic Control-Flow Graph

Figure 21: Translation of a `while` loop

$$
\begin{aligned}
\mathit{SimpleStmt} ::= \ & \textbf{assert}\ \mathit{Expr} \mid \textbf{assume}\ \mathit{Expr} \mid \textbf{exprStmt}\ \mathit{Expr} \\
\mid \ & \textbf{goto}\ \mathit{blockId} \mid \textbf{havoc}\ \mathit{l\text{-}value} \\
\mid \ & \textbf{sequence}\ \mathit{SimpleStmt}\ \mathit{SimpleStmt} \\
\mid \ & \textbf{varDecl}\ \mathit{name}\ \mathit{type}
\end{aligned}
$$

Figure 22: Desugared-Statement Language

$$
\begin{aligned}
\mathit{CfgStmt} ::= \ & \textbf{assert}\ \mathit{Expr} \mid \textbf{assume}\ \mathit{Expr} \\
\mid \ & \textbf{goto}\ \mathit{blockId} \mid \textbf{sequence}\ \mathit{CfgStmt}\ \mathit{CfgStmt} \\
\mid \ & \textbf{varDecl}\ \mathit{name}\ \mathit{type}
\end{aligned}
$$

Figure 23: Final CFG Language

4.1.2.4 Final Desugaring

ESC/Java2 makes heavy use of compiler-option specific desugarings to delay as long as possible the decision of how to translate certain constructs, depending on how the user has configured the verification session [Leino *et al.*, 1998]. Currently no desugarings of this kind are performed in ESC4, but we leave open the possibility that future enhancements of this sort will be wanted. A grammar for the fully desugared language is shown in Figure 22.

4.1.2.5 Passification

The final step in producing the CFG program from a method is to remove all side effects. A grammar for the fully desugared, acyclic, passive, CFG-statement language is shown in Figure 22. Unlike Java expressions, the CFG's *Expr* are not allowed to have side effects, so all assignments and method calls are replaced with passive versions that contain only assumes and asserts. This is done by transforming it to a Dynamic Single Assignment (DSA) form.

Passification to DSA is also discussed briefly in [Barnett and Leino, 2005], where an optimization is given that in some cases reduces the number of incarnations needed for some l-values. (In this subsection we use the term *l-values* to refer to all kinds of expressions that can appear on the left-hand side of an assignment expression, such as local variables, field references, and array accesses)

The input to the passification stage is an acyclic CFG program, which consists of a set of blocks and the name of the start block. Blocks have a name, a (possibly compound) statement, and a list of following blocks. To make processing easier, we determine for each block a set of parent and children blocks. Blocks are also augmented with an *incarnation map*, which stores a mapping from assignables (i.e., variables, fields, and arrays) to the set of integers representing the incarnations with that assignable's latest value. A set is used so that multiple incarnations can be allowed to have the latest value, and this extra information can be used to reduce the number of incarnations and extra synthetic assumptions needed when reconciling blocks.

The first step in passification is performing a topological sort (see, e.g., [Cormen *et al.*, 1990, §23.4]) of the CFG program's blocks. This produces an ordering of the nodes in which a block comes before any of its children. We then iterate through the sorted list of blocks performing the following three steps:

1. If the current block has more than a single parent then reconcile the parents' incarnation maps.

2. Set the information in the current block's incarnation map to that of its parents.

3. Perform a DSA transformation on each statement in the current block.

We now look at each of these in more detail.

Reconciling the parents' incarnation maps is a more step. First, we must find all of the l-values mentioned in the current block's parents' blocks. We then iterate over this set, vs, of l-values. For each l-value, if the parent blocks share a common incarnation then that incarnation is used. Otherwise, we must create a new incarnation that is larger than any parent's incarnation for that l-value and update each of the parent blocks by

1. adding to each parent an assumption that the l-value with the new incarnation is equal to that parent's largest incarnation and

2. adding the new incarnation for the l-value to the parent's incarnation map.

Setting the information in the current block's incarnation map to that of its parents means setting the incarnation of each l-value in vs to the largest incarnation mentioned in the intersection of the incarnation sets from the parents for this l-value. Because of the reconciliation of the parents' maps, this intersection will not be empty. If a block has no parents, which is the case for the starting block and unreachable blocks, then its incarnation map is left empty by these first two steps.

Transformation a statement to DSA form gives an explicit incarnation to all variable references, and all assignments of the form

$$x := E$$

are replaced with `assume` statements of the form

$$\texttt{assume}\ x_k == E,$$

where k is an explicit incarnation that is larger than any incarnation x may have had *after* processing E. If the assignment expression's subexpressions have embedded assignments, these are passified and included as assumptions before the

statement currently being visited. This is done by storing a side-effect list, which stores the assumptions (and assertions, as we shall see below when discussing method calls) that need to be made before the current statement.

As an example, consider an assignment expression with an incarnation map with i and k both mapped to $\{0\}$.

```
i = (i++) * (--k) / m()
```

If the called method `m()` does not have an `assignable` clause then we translate this expression to the following:

$$\texttt{assume } i_1 = i_0 + 1$$

$$\texttt{assume } k_1 = k_0 - 1$$

$$\texttt{assert } m_{precondition}$$

$$\texttt{assume } m_{postcondition}$$

$$\texttt{assume } i_2 = i_0 * k_1 \ / \ m_{result}$$

Method calls are replaced with a new variable holding the result. The declarations and initializations of new binding variables to hold the evaluation of the actual parameters are added to the side-effect list, followed by the assertion of the method's precondition (and any necessary invariants) with the method's formal parameters replaced with the new binding variables. This substitution is performed by a `SimpleSubstVisitor`. Any variables mentioned in the called method's assignable clause have their incarnation replaced with a new, larger one before processing the post condition. This havocking of assignable targets has the effect of allowing the modified variables to take on any values, restrained only by the method's postcondition. The assumption of the invariants and the postcondition are similarly added to the side-effect list. Note that it is not necessary for there

to be an explicit assignment to the result's variable, since the only knowledge we can have about its value comes from assuming the method's postcondition.

ESC4 supports both JML's logical and arithmetic quantified expressions. The logical quantifiers are the universal and existential, and the arithmetic quantifiers include `sum`, `min`, `max`, and `number_of`. Declarations for these expressions' bound variables are added to the side-effect list. Before passifying the range and body of the quantified expression, the contents of the side-effect list are stored in a local variable, and the list is emptied. The range and body's side effects, which can only be caused by method calls, are formed into an expression that is made to imply the body, giving a new body. The side-effect list is restored before returning the result of passifying the quantified expression, which is itself a quantified expression. In the case of the universal quantifier, the range is made to imply the new body, while for the existential, the range is conjoined to it.

Conditional expressions (i.e., those with the form $a\,?\,b : c$) requires a bit more work than some other expressions. After passifying each of its three parts, the side-effect list is stored to local variables. Copies of the incarnation map that results from passifying the condition are made so that identical maps can be used to passify the two alternative expressions. If either of these passification steps detects side effects, the incarnation maps must be reconciled, similar to what is done when joining two parent blocks. This reconciliation ensures that the maximum incarnation for each assignable on both branches is the same.

4.1.2.6 Final words on Generating the CFG Program

The steps in this subsection described the transformation of an AST to a desugared, passified, acyclic Control-Flow Graph program. The only statements that

$$wp(\texttt{assert}\ P, Q) = P \wedge Q$$
$$wp(\texttt{assume}\ P, Q) = P \longrightarrow Q$$
$$wp(S;T, Q) = wp(S, wp(T, Q))$$

Figure 24: Weakest precondition for passive statements

remain in the language are for variable declarations, assertions, assumptions, sequencing, and gotos. The following will discuss the transformation to a Verification Condition.

4.1.3 VC Generation

The Verification-Condition Generator converts a CFG program to a VC. In comparison to the AST language, the CFG language processed in this stage is minuscule. Using a weakest-precondition calculus, we compute a VC for each method. The conversion provided in [Barnett and Leino, 2005] is reproduced in Figure 24. Note that the infix simicolon (';') is used to form the sequence of two statements.

For each block with label A and body S, an auxiliary variable is introduced A_{ok} with the form

$$A_{ok} \equiv wp(S, \bigwedge_{B \in Succ(A)} B_{ok})$$

. Let us call this block equation definition A_{be}.

The VC for a method is formed from the conjunction of these definitions implying the auxiliary variable for the starting block. To enable the most flexibility for the Theorem-Prover Back End, we leave the VC in this VC Program form. The VC Program, or list of block equations, for our running example is shown in Figure 25. The VC is then

$$(\text{start\$be} \wedge \text{else\$1\$be} \wedge \text{then\$1\$be} \wedge \text{return\$be}) \longrightarrow \text{start\$ok}.$$

In the next section we see how ESC4 discharges VCs once they are generated.

start: return@0$0 :: **int**;
x@122$0 :: **int**;
(True∧True)⟶(then1ok∧(else1ok∧True)))
else$1: (¬ (x@122$0 ≥ 0))⟶((return@0$0 == x@122$0)⟶(return$ok∧True))
then$1: (x@122$0 ≥ 0)⟶((return@0$0 == x@122$0)⟶(return$ok∧True))
return: ((True∧((return@0$0 ≥ 0)∧
((return@0$0 == x@122$0) ? True : (return@0$0 == −x@122$0))))∧True)

Figure 25: VC Program for `abs`

4.2 Discharging VCs

> "Logic and mathematics seem to be the only domains where self-evidence manages to rise above triviality; and this it does, in those domains, by a linking of self-evidence on to self-evidence in the chain reaction known as proof." — Willard van Orman Quine

In the previous section we saw how ESC4 generates VCs from the AST it takes as input. In this section we will see how those VCs are discharged and failures reported to the user.

4.2.1 Prover back-end

A class diagram for the Prover back-end is shown in Figure 26. A configurable Prover Coordinator is used to discharge VCs. It obtains a proof strategy from a factory whose behavior is governed by compiler options. The default strategy is a sequence of two strategies: The first tries to prove the entire VC using a single Automated Theorem Prover (ATP). If it fails, the second, `ProveVcPiecewise`, is used. Both use adapters to access the theorem provers. These adapters hide the mechanism used to communicate with the provers. They use visitors to pretty print the

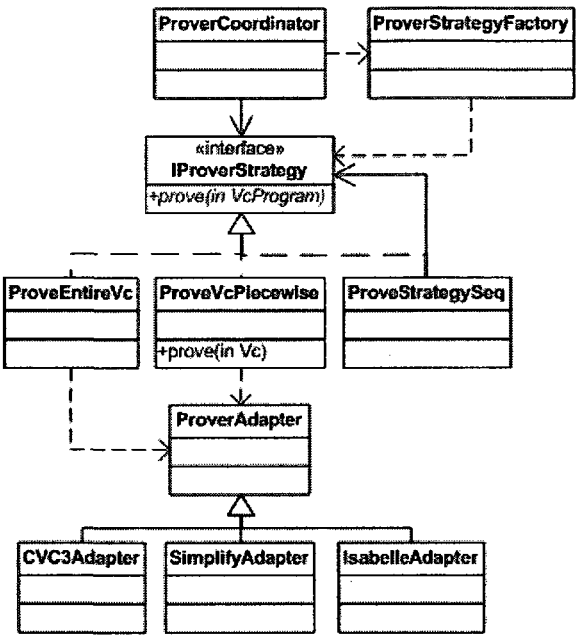

Figure 26: ESC4's prover back-end

VC to produce input for each ATP's native language. To eliminate wasting time re-discharging a previously discharged VC (or sub-VC), the strategies can make use of a VC cache, which is persisted. A Universal Background Predicate (UBP) is needed for each of the provers being used. A UBP is a collection of definitions that provide the semantics of Java and JML.

`ProveVcPiecewise` implements 2D VC Cascading: VCs are broken down into sub-VCs, giving one axis of this 2D technique, and proofs are attempted for each sub-VC using each of the supported ATPs, giving the second axis [James and Chalin, 2009c].

The conjunction of the set of sub-VCs is equivalent to the original VC. Discharging all of the sub-VCs shows that the method is correct with respect to its specification. Any sub-VCs that cannot be discharged reflect either limitations of the provers or faults in the source.

The splitting is done by recognizing that VCs are sequences of implications and conjunctions in which the atomic conjuncts are (usually) generated from assertions in the CFG program and implications from assumptions. The implications are distributed over the conjunctions to form a set of implications whose conjunction is equivalent to the original VC. For example, a VC of the form

$$a \longrightarrow (b \& c)$$

is converted to the set

$$\{a \longrightarrow b, a \longrightarrow c\}.$$

Each of the resulting sub-VCs represents a single acyclic path from the method's precondition, through its implementation to an assertion. An Isabelle/HOL proof that this decomposition is sound and complete can be found in Appendix A. Next we remove the sub-VCs that end in $\longrightarrow True$, since these are trivially valid. As a final step, we replace all but the final implication with conjunctions: i.e., we convert sub-VCs of the form

$$(a \longrightarrow b) \longrightarrow c$$

to

$$(a \wedge b) \longrightarrow c$$

Instead of reporting an entire assertion as failing, we try to identify the smallest possible responsible subexpression. This helps users more quickly locate and correct the problem. To do this, we split not only conjunctions generated during translation and VC generation but also the conjunctions in the source code. Initially, we naïvely split conditional conjunctions (`&&`) in the same way as logical

(i.e., non-short circuiting) conjunctions (`&`), but this led to incorrect error reporting. For example, consider the assertion

```
assert x == 3 && y == x + 3.
```

The second conjunct should never be reported as unprovable if the first conjunct is unprovable. Since the second conjunct should only be reported as unprovable if the first conjunct is true, the second conjunct should be prefixed with this implication. Thus, the conditional conjunction should be translated as

$$a \,\&\&\, b \equiv a \,\&\, (a \longrightarrow b).$$

Once the method's VC has been split into sub-VCs, we try to show that it is valid by passing it in turn to Simplify, CVC3 and Isabelle/HOL. By far, Simplify is the fastest of these three, when it is able to discharge a VC, and it is the first that ESC4 uses. Before invoking Isabelle, we try a novel technique: We negate the sub-VC's consequent and try to validate this new sub-VC using Simplify. That is, if the original sub-VC has the form

$$\bigwedge A_i \longrightarrow c$$

then the new sub-VC has the form

$$\bigwedge A_i \longrightarrow \neg c.$$

If successful, we know not only that the original sub-VC is invalid, but also that this invalidity is not due to a lack of power on the part of the ATPs used. This extra information is passed to the user. This is usually a faster operation than invoking Isabelle, and since it is easy for programmers to write the negation of

the expression wanted, it is useful to know when it occurs. Only after all other attempts fail is Isabelle invoked.

Isabelle, which is more commonly used as an interactive theorem prover, is used as an ATP by having it use a hard-coded proof strategy[3]. Isabelle is much slower than the other two provers, but this is compensated for by its being able to discharge many VCs that other ATPs cannot. Section 5.2 describes another way that the power of Isabelle is used in ESC4.

4.2.2 Returning to our example

In this subsection we look at how ESC4 produces the warnings shown in Figure 15 from the VC program we derived in the previous section (see Figure 25).

As mentioned above, the default strategy used by the Prover Coordinator is to first try to verify the entire VC program. Since there is a known problem with the code, we are not surprised that Simplify is unable to verify it.

Splitting the VC program produces 4 sub-VCs, all with `return` and `x` declared to be of type `int`:

1. $\bigwedge$ ((True $\wedge$ True), ($\neg$ (x@122$0 $\geq$ 0)), (return@0$0 == x@122$0))

   $\longrightarrow$ (return@0$0 == x@122$0) ? True : (return@0$0 == (- x@122$0))

2. $\bigwedge$ ((True $\wedge$ True), ($\neg$ (x@122$0 $\geq$ 0)), (return@0$0 == x@122$0))

   $\longrightarrow$ (return@0$0 $\geq$ 0)

3. $\bigwedge$ ((True $\wedge$ True), (x@122$0 $\geq$ 0), (return@0$0 == x@122$0))

   $\longrightarrow$ ((return@0$0 == x@122$0) ? True : (return@0$0 == (- x@122$0))

4. $\bigwedge$ ((True $\wedge$ True), (x@122$0 $\geq$ 0), (return@0$0 == x@122$0))

   $\longrightarrow$ (return@0$0 $\geq$ 0)

[3]For now, "by (simp add: nat_number | auto | algebra)+" is used.

```
(IMPLIES (AND (AND (EQ |@true| |@true|)
                   (EQ |@true| |@true|) )
              (NOT (EQ |@true| (integralGE |x@122$0| 0)) )
              (EQ |@true| (integralEQ |return@0$0| |x@122$0|)))
         (LBLNEG |Postcondition@114| (LBLNEG |eq@37_48|
         (EQ |@true| (integralGE |return@0$0| 0)))))
```

Figure 27: Simplify encoding of the problem sub-VC

All of these are provable by Simplify except the second. For this sub-VC, the Simplify source generated is shown in Figure 27. It is an implication in which the antecedent is a conjunction of the assumptions, and the consequent, a post-condition, is an assertion. Simplify outputs that this is invalid and reports the labels `|eq@37_48|` and `|Postcondition@114|`. If no other ATP is able to discharge this sub-VC, these will be used to produce an error or warning to the user that the Precondition that starts at source position 114 may not hold because of the expression at position 37–48. Variables are not declared, and integral comparisons are made with predicates defined in the Simplify UBP, which we take wholly from the ESC/Java2 distribution. `LBLNEG` introduces a label that is output if the enclosing expression contributes to the whole expression not being verifiable.

After Simplify is seen to fail, CVC3 is tried. The much more human-readable CVC3 source generated for the problem sub-CV is shown in Figure 28. For CVC3, the variables must be declared before their use, and the sub-VC is presented as a query. Integral comparisons are native operations.

After CVC3 is seen to fail, we try to show the negation of the sub-VC using Simplify. The Simplify source is identical to that shown in Figure 27 except that the `Postcondition` label is wrapped in "`(LBLNEG |not@37_48| (NOT ...))`." Since Simplify

```
return$0$0 :  INT ;
x$122$0 :  INT ;


QUERY ( (((TRUE AND TRUE) AND
          (NOT (x$122$0 >= 0)) AND
          (return$0$0 = x$122$0)
         ) => (return$0$0 >= 0)) );
```

Figure 28: CVC3 encoding of the problem sub-VC

```
theory Thesis_abs_2 imports ESC4 begin
lemma main:  ''([| (True & True); (~ ((x_122_0::int) >= (0::int)));
                 ((return_0_0::int) = (x_122_0::int))|]
               ⟹ ((return_0_0::int) >= (0::int)))''
by (simp add: nat_number | auto | algebra)+
```

Figure 29: Isabelle encoding of the problem sub-VC

is able to show this negated version to be valid, the original sub-VC is provably false, and this is indicated in the second line of the error shown in Figure 15.

Since the truth value of the sub-VC has been determined, the `ProveVcPiecewise` strategy will not invoke Isabelle, but for the sake of illustration, we continue on. The Isabelle/HOL source generated for the problem sub-CV is shown in Figure 29. Variables do not need to be declared, but all terms are given with their types, even litterals. The UBP is stored in the `ESC4` theory, which also imports the theory `Main`. The double square brackets ([| ... |]) expression can be read as the conjunction of its semicolon-separated subexpressions.

4.2.3 Reducing Prover Invocations

Isabelle's power comes at the cost of it being slower than the other ATPs used. It is not uncommon for it to take 10 times longer than Simplify to process a VC, but it is able to discharge whole classes of VCs that Simplify cannot. Even though the other ATPs are faster than Isabelle, they are much slower than simple manipulations of in-memory data structures or simple checks of the file system. ESC4 uses several techniques to help offset the theorem provers' cost by eliminating unnecessary invocations of them.

4.2.3.1 Caching

Since ESC4 is run every time that a method is saved and successfully compiled, it is important that it be as quick as possible. To help with this goal and to eliminate redundant calls to the theorem provers, once a VC has been proven, it is stored in a persisted cache. Before sending a VC to any of the ATPs, the system checks if the VC cache already contains it. If so, it is discharged immediately. If it is not found but a prover is able to show that it holds true, then it is added to the cache. Isabelle is currently the last prover in our prover chain. If it is not able to discharge a VC then some information is left in the file system that indicates this situation. If this indication is present, then none of the theorem provers is able to prove it, and we can immediately return this failure status.

The cache stores the text of the VC as a `HashSet`. The cache is stored to the file system on a per-compilation-unit basis. Since there are relatively few VCs in each instance of the cache, the lookup time is insignificant. This cache is consulted before calling any of the ATPs. Also, the ProverCoordinator leaves some

information in the file system so that it can determine whether Isabelle was previously unable to discharge a VC. This eliminates invocations of Isabelle that are known will fail.

4.2.3.2 A More Robust Cache

VCs are fragile with respect to source-code edits. Information about expressions' source-code positions is added to identifiers in the generated VCs, and this position information is used for two purposes: for error reporting and for making identifiers unique. Unfortunately, having position information in the VCs is a major source of brittleness of both the VC cache and the Offline User Assisted ESC (OUA-ESC) process [James and Chalin, 2009c]. With it, adding even a single character to the source file would cause the text of the cache entry or generated lemma to change. To avoid this, we plan to remove position information whenever possible from lemmas in both the VC cache and the lemmas sent to Isabelle. This will not cause a problem with error reporting because only VCs that are true are stored in the cache and because we use the problems that are indicated by Simplify to provide error reporting. Making identifiers unique, can be partially addressed by only including the position information if the same identifier is used more than once in a given sub-VC (e.g., if two quantifiers' bound variables share the same name). A further optimization would be to replace an absolute position with a relative position (so, e.g., the two aforementioned bound variables would be suffixed with _1 and _2 instead of their character positions).

Another novel technique is used to keep from having Isabelle waste time trying to discharge a VC that is easily proved false. Before invoking Isabelle, a faster ATP is used to try to prove its negation (or rather, the negation of the original assert). For example, if the original VC has the form $(p \longrightarrow q)$ then ESC4 tries to

show $(p \longrightarrow \neg q)$. If this modified VC can be shown to be true then the original VC must be false[4], and this extra information can be reported to the user. It is often useful to know that an assertion *is* false rather than just that the theorem prover was unable to prove it true.

4.2.4 Post Processing Results

Once the Prover Coordinator has finished processing a VC program, it returns a result, which can be either "valid" or information about a specification violation. These latter include

1. the kind of assertion that failed (e.g., in-line assertion or postcondition),
2. the source starting and ending positions of the offending assertion expression,
3. the source starting and ending position the failure was detected (not always present),
4. the name of the sub-VC, which we will see a use for in Section 5.2, and
5. an indication if the sub-VC was proved false.

In this chapter we have seen how ESC4 processes the AST provided to it by JML4 to verify code and alert the user of specification violations. In the next chapter, we will look at some examples of code that ESC4 is able to verify that other static-analysis tools cannot.

[4]or contain a contradiction, which would mean either the specification introduced a contradiction or the assertion corresponding to the VC is unreachable.

Chapter 5

ESC Enhancements

In this chapter we examine some of the enhancements that allow ESC4 to verifiy code that similar tools cannot. Section 5.1 provides some of the benefits of multi-prover support. Section 5.2 presents Offline User-Assisted ESC.

5.1 Enhanced ESC in ESC4[1]

"Take the best that exists and make it better." — Henry Royce

5.1.1 Overview

Extended Static Checking (ESC) tools, such as ESC/Java2 [Cok and Kiniry, 2005], provide a simple-to-use, compiler-like interface that points out common programming errors by automatically checking the implementation of a class against its specification. Even though ESC/Java2 is the de facto ESC tool for the Java Modeling Language (JML) [Leavens *et al.*, 2008], it has some important limitations. For example, it is unable to verify the assertions shown in Figure 30, which contain

[1]This section is based on [James and Chalin, 2009a].

1. numeric quantifiers,

2. quantified expressions in certain positions, and

3. non-linear arithmetic.

```
assert 6 == (\product int i; 0 < i && i < 4; i);
assert (E1 ? E2 : (\forall int i; i > 0; i != 0));
maintaining b == 3*(x-c)*(x-c);
```

Figure 30: Assertions that ESC/Java2 cannot verify

In fact, the second of these limitations, as well as others, exists even in more modern ESC tools like Spec#'s Boogie program verifier [Barnett *et al.*, 2005]. We believe that ESC4 is the first ESC tool that can automatically verify all of these.

In this section we report on work done to overcome these limitations in ESC4. In the next subsection we report some examples of methods that ESC4 is able to verify but that ESC/Java2 cannot. Related Work is described in Section 5.1.3.

5.1.2 ESC4 Enhancements

In this subsection we describe some of the enhancements made to ESC4. In general, we do so by presenting examples of specifications that ESC/Java2 is unable to verify[2] but that ESC4 can handle. The mechanisms by which the verification is made possible are also described. In particular, we explain the following enhancements: support for

- numeric quantifiers,

- quantified expressions anywhere a boolean expression is allowed, and

[2]Most are also beyond the capabilities of Boogie, as will be explained in Section 5.1.3.

- non-linear arithmetic.

In addition, we also briefly comment on ESC4's ability to report assertions that are provably false.

5.1.2.1 Arithmetic quantifiers

Besides existential and universal quantifiers, JML supports the generalized numeric quantifiers `\sum`, `\product`, `\min`, `\max`, and `\num_of`. Like the logical quantifiers, these have one or more bound variables, an optional expression limiting the range of these variables, and a body expression. An operator is folded into all values the body expression can take on when the range expression is satisfied. As expected, the expression

```
(\sum int i; 3 < i & i < 7; i)
```

evaluates to 15 (i.e., $4 + 5 + 6$).

ESC/Java2 uses Simplify as its underlying Automated Theorem Prover (ATP). ESC4, however, makes simultaneous use of a range of ATPs, currently Simplify, CVC3, and Isabelle/HOL. ESC/Java2 translates numeric quantified expressions as uninterpreted constants since Simplify is unable to cope with them. ESC4 does so as well for Simplify and CVC3, but since Isabelle is able to work with expressions in higher-order logic, ESC4 faithfully translates all quantified expressions for it. As a result, many methods that use them can be verified automatically.

Figure 31 shows how compactly the specification for the factorial function can be expressed. Without numeric quantifiers it would be difficult[3] to express such a contract.

[3] If not impossible, given that it is unclear what the semantics of recursive method contracts are in JML.

```
//@ requires n >= 0;
//@ ensures \result == (\product int i; 1 <= i & i <= n; i);
public static int factorial(int n) {
    int result = 1, j = 1;
    //@ maintaining result == (\product int i; 1 <= i & i <= j-1; i);
    //@ maintaining 1 <= j;
    //@ decreases   n-j+1;
    while (j != n+1)
        result *= j++;
    return result;
}
```

Figure 31: Arithmetic quantified expression

Numeric quantified expressions are translated into one of two forms, depending on the syntactic form of the range. If explicit bounds can be determined, then the expression is translated into an Isabelle function call that is very amenable to use in automatic verification. For example, the JML expression

$$(\backslash\texttt{sum}\ \textbf{int}\ \mathrm{i};\ a <= \mathrm{i}\ \&\ \mathrm{i} <= b;\ E(\mathrm{i}))$$

would be translated into Isabelle as

$$\mathrm{sum}\ a\ b\ (\lambda \mathrm{i}.E(\mathrm{i}))$$

The Universal Background Predicates (UBPs) [Flanagan *et al.*, 2002] in ESC4 are prover-specific collections of definitions that provide the semantics of Java and JML. The definition of "sum," the summation function from the UBP for Isabelle, is shown in Figure 32. This UBP also contains some lemmas for dealing with -1 and **nats**, which are needed e.g., in loop invariants. Note that the range is shown as having type **int** $\Rightarrow$ **int**, which is a function from **int** to **int**. This function is

```
fun sum_helper ::
    "nat ⇒ int ⇒ (int ⇒ int) ⇒ int"
where
  "sum_helper 0       lo body = 0"
| "sum_helper (Suc n) lo body =
   (body (int n + lo)) + (sum_helper n lo body)"

fun sum :: "int ⇒ int ⇒ (int ⇒ int) ⇒ int"
where
"sum lo hi body =
      sum_helper (nat (hi - lo + 1)) lo body"
```

Figure 32: Definition of "sum" from Isabelle UBP

formed as a lambda expression whose single bound variable is that of the quantified expression.

When ESC4 cannot determine a numeric range, Isabelle's set-comprehension notation [Nipkow *et al.*, 2002, §6.1.2] is used, which allows the translation to capture the full meaning of the original JML expression. Hence, the JML expression

$$(\backslash\texttt{sum}\ \textbf{int}\ \mathrm{i};\ R(\mathrm{i});\ E(\mathrm{i}))$$

would be expressed as the Isabelle expression

$$\Sigma\ \{\ E(\mathrm{i})\ |\ \mathrm{i}\ .\ R(\mathrm{i})\ \}$$

While lemmas containing the set-comprehension form are not easily discharged automatically, Section 5.2 describes how verification conditions (VCs) can be manually discharged.

5.1.2.2 Restoring First-Class Status of Quantified Expressions

As shown in the second assertion in Figure 30, ESC4 allows quantified expressions to appear in conditional expressions. Simplify makes a strong distinction between formulas and terms, and permitting quantifiers only as formulas. The implementation in Simplify's UBP of conditional expressions, including conditional conjunctions and disjunctions (i.e., && and —), requires that their subexpressions be terms. ESC4 uses an only slightly modified version of ESC/Java2's UBP for Simplify [Flanagan *et al.*, 2002], so its use of Simplify has the same restriction. The other ATPs used by ESC4 do not have this limitation, as their input languages provide support for conditional expressions. This permits ESC4 to treat quantified expressions as first-class expressions.

5.1.2.3 Non-linear arithmetic

Figure 33 shows a JML-annotated method that computes the cube of its integer parameter using only shifts (multiplication by 2) and additions [Kolman and Busby, 1986]. This method is an interesting example for verification because it is far from obvious that its body respects its simple contract.

Without adequate tool support for static verification, extensive testing would be needed to build confidence in the method's correctness. ESC/Java2 is unable to verify this method because its underlying ATP, Simplify, is unable to reason about non-linear arithmetic. This is true of most ATPs. Instead of relying on a single theorem prover, ESC4 simultaneously uses a range of theorem provers (see Section 4.2). For our Cube example, Simplify is able to discharge all of the VCs except for the ones corresponding to the last two loop invariants, which Isabelle is able to discharge automatically. Since all of the VCs can be discharged by at least one of the ATPs, ESC4 is able to verify that the method is correct.

```
//@ requires x > 0;
//@ ensures \result == x * x * x;
public int cube(int x) {
   int a = 1,
 b = 0;
   int c = x,
 z = 0;
   //@ maintaining a == 3*(x-c) + 1;
   //@ maintaining b == 3*(x-c)*(x-c);
   //@ maintaining z == (x-c)*(x-c)*(x-c);
   //@ decreasing c;
   while (c > 0) {
         z += a + b;
         b += 2*a + 1;
         a += 3;
         c--;
   }
   return z;
}
```

Figure 33: Computing x^3 with shifts and additions

5.1.3 Related Work

5.1.3.1 ESC/Java and ESC/Java2

ESC/Java2 [Cok and Kiniry, 2005] is the successor to the earlier ESC/Java project [Flanagan *et al.*, 2002], the first ESC tool for Java. ESC/Java's goal was to provide a fully automated tool to point out common programming errors. The cost of being fully automated and user friendly required that it be—by design—neither sound nor complete. Soundness was lost by not checking for some kinds of errors (e.g., arithmetic overflow of the integral types is not modeled because it would have required what was felt to be an excessive annotation burden on its users). ESC/-Java provides a compiler-like interface, but instead of translating the source code to an executable form, it transforms each method in a Java class to a VC that is

checked by an ATP. Reported errors indicate potential runtime exceptions or violations of the code's specification. "The front end produces abstract syntax trees (ASTs) as well as a type-specific background predicate for each class whose routines are to be checked. The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that routines in that class use" [Flanagan *et al.*, 2002]. The ESC/Java2 project first unified the original program's input language with JML before becoming the platform developed by many research groups.

5.1.3.2 Spec#, VCC, and HOL-Boogie

Spec# is Microsoft's extension to C# for supporting verified software [Barnett *et al.*, 2005]. It is composed of

- the Spec# programming language, a superset of C# enriching it with support for Design by Contract
- the Spec# compiler, which includes an annotated library, and
- the Boogie static verifier, which performs ESC.

The Spec# system is among the most advanced ESC tools currently available. A key developer of Spec#, K. Rustan M. Leino, is one of the original developers of ESC/Java, so it is not surprising that the lessons learned from that experience were put to use in developing Spec#.

We translated the JML code in Sections 5.1.1 and 5.1.2 into Spec# and tested them with version 1.0.20411.0 (11 April 2008) under Visual Studio 2008. We were surprised at the results. Of the three assertions in Figure 30, only the first was correctly handled, while the second caused the IDE to throw an exception. When

processing the example that showed ESC4's ability to indicate that a subexpression is provably false, Spec# is only able to detect that the assertion is violated. That is, it is unable to identify the offending subexpression or to indicate that the expression is definitely false.

Leino and Monahan [Leino and Monahan, 2007] report a way to handle arithmetic quantifiers using ATPs. This addition was enough to allow the simple example in the first assertion in Figure 30 to be verified, but not the example in Figure 31.

Böhme, Leino, and Wolff [Böhme *et al.*, 2008] report on HOL-Boogie, an extension of Isabelle/HOL that can be used in place of the Z3 ATP in the regular Boogie toolchain. Currently it is used in the VCC toolchain, but the Spec# system could be modified to make use of it. This addition would have the possibility of allowing Spec# to verify most of the examples presented in this section, although the verification would require manual proofs, as there is no indication that Isabelle is used as an ATP.

HOL-Boogie does not provide proof-status feedback to the IDE (i.e., VisualStudio). Our approach does provide such feedback, with the goal of being able to do the proofs within the JML4 IVE, thus delivering a more satisfying user experience.

Like ESC4, HOL-Boogie supports splitting a method's VC into sub-VCs, which it then tries to discharge using Z3 and Isabelle/HOL. Unlike ESC4, this splitting is done by Isabelle, which itself makes calls to Z3. Any user-supplied proofs are *not* used to discharge the corresponding sub-VCs.

5.1.3.3 Krakatoa and Caduceus

Krakatoa [kra, 2009] is an FSPV tool for JML-annotated Java classes. Originally designed to generate theories for the Coq theorem proven it has recently being modified to output programs for the Why tool [Filliâtre and Marché, 2007]. Caduceus [Filliâtre *et al.*, 2008] is similar to Krakatoa, but it verifies C programs annotated with contracts similar to those of JML. Verification in Krakatoa or Caduceus is essentially a three-step, manual process: C programs are first translated into the language of the Why system [Filliâtre, 2008], then Why is used to translate VCs into the language of a user-selected prover.

Why is multi-tool Verification Condition (VC) generator. The input syntax of Why is a Why program. A Why program may contain assignment, loop, and conditional statements, as well as function declarations. Additionally it supports throwing and catching exceptions and has limited support for expressions with side-effects. It supports annotations for function declarations and loop statements.

The Why tool transforms input programs into VCs using a weakest precondition semantics proven sound using the pen and paper approach [Filliâtre, 2003]. The output is one or more theories for a number of provers. These include the automated Simplify, Z3, Yices, and CVC3; and the interactive Coq, Isabelle, and PVS.

Finally, the user runs or interacts with the selected prover inorder to discharge the VC proof obligations. The user is left to interpret any prover output, including tracking undischarged VCs back to the source. Such an *offline* approach to verification is like that adopted by the JML4 FSPV Theory Generator [Chalin *et al.*, 2008a, Karabotsos *et al.*, 2008] and contrasts with ESC4's fully automated mode of extended static checking.

Like ESC4, Caduceus treats quantified expressions as first-class expressions. In addition to function calls being allowed in specifications, a construct called *predicates* allows the definition of specification-only functions.

The ATPs used by the Why tool cannot reason about numeric quantifiers or non-linear arithmetic, so code that makes use of them could not be verified automatically. The interactive prover would allow them to be proved manually.

5.1.3.4 SPARK

The SPARK toolset [Barnes, 2006] also provides support for discharging VCs using both a fully automatic and an interactive prover. A report generator is used to combine the results of the verification process to provide information about the status of a program's verification. All of these tools are stand-alone and are invoked from the command-line. (See also Appendix C.)

5.1.4 Summary

In this section we presented several examples of code that ESC4 is able to verify and that other commonly used ESC tools are not. ESC4 can verify code that uses numeric quantifiers. Unlike other ESC tools, ESC4 does not limit quantified expressions to being the only expression in an assertion. Also, we believe that ESC4 is the first ESC tool for JML that can verify code that uses non-linear arithmetic. It is able to do these things because it uses a variety of ATPs, where one is able to compensate for the weaknesses of the others. VCs that can be proved to never hold are reported as such to users. Those that are proved true are cached to eliminate unnecessary invocations of the theorem provers.

ESC4 is a quickly evolving research platform. Even though there are things it can do that ESC/Java2 cannot, there is much more that ESC/Java2 can do that

ESC4 does not yet do. To close this gap, we are continuing to flesh out ESC4's capabilities to more fully support Java and JML.

In the next section, we present Offline User-Assisted ESC, which we believe is another powerful addition to static verification.

5.2 Offline User-Assisted Extended Static Checking[4]

> "The moment we want to believe something, we suddenly see all the arguments for it, and become blind to the arguments against it." — George Bernard Shaw

In addition to overcoming the limitations described in the previous section, ESC4 has other enhancements. Most notably, it introduces a new category of static verification, called Offline User-Assisted ESC (OUA-ESC), that falls between the fully automated classical ESC and interactive Full Static Program Verification (FSPV).

Offline User-Assisted ESC (OUA-ESC) is a novel form of static verification that lies between the fully automatic classical ESC (which is incomplete) and interactive and complete FSPV. OUA-ESC enables a developer to take advantage of the full power of Isabelle as an interactive theorem prover to discharge a VC that cannot be discharged automatically. Once the proof has been written, ESC4 makes use of it during subsequent compilation cycles, enabling Isabelle to act as an ATP over the user-supplied proof. Hence, OUA-ESC allows JML4's static verifier to take advantage of the full power of Isabelle—in conjunction with user-supplied proofs—as one of its automatic theorem provers, thus increasing the ESC4's completeness to the same level achievable by FSPV.

[4]This section is based on [James and Chalin, 2009c].

To do so, skeleton Isabelle theory files are created for any VCs that Isabelle is unable to discharge. If the user provides a valid proof for the lemmas in those files then the next compilation cycle's invocation of ESC4 will use the provided proof instead of the default proof strategy. If Isabelle is able to prove the lemma then the VC is taken as discharged. OUA-ESC opens up the possibility of verifying many more methods than would be possible using only ATPs, and without forcing users to provide proofs if they are not so inclined.

With the addition of this ability to make use of arbitrarily complex proof techniques, ESC4 is able to discharge any VCs that are produced, limited only by the capabilities of Isabelle and the skills and needs of the user. If the user does not want to manually discharge a VC, the lemma file can still prove useful, as it contains a trace of the method from the precondition through the body to an assertion that is reported as not holding.

Isabelle's automatic simplification commands, while not enough to prove the lemma, are usually able to reduce the original, quite large, lemma to subgoals that show only the missing facts that would allow the proof to go through. Often these smaller forms are

- obviously true and just require a little manipulation for Isabelle to recognize their truth,
- obviously false and lead to a search in the VC for clues to the error in the source code, or
- surprisingly false and lead to the modification of specifications to allow the necessary information to be available.

The last case is most apparent when assumptions are missing, such as method preconditions or loop invariants.

Both of the automatic Isabelle commands **quickcheck** and **refute** can provide counterexamples that are often helpful in determining where the code or specification is incorrect by pointing out why Isabelle thinks the lemma is false. Learning just a little bit about Isabelle and ProofGeneral is enough to allow the gathering of a lot of useful information about an undischarged VC. In addition to the normal simplification procedures, the 2008 release of Isabelle [isa, 2008] includes the **sledgehammer** command [Paulson and Susanto, 2007], which can automatically search for some slightly more sophisticated proofs. When the **sledgehammer** finds a proof, a proof script is provided that can be used as a user-supplied proof.

Once a proof for a VC has been accepted by Isabelle, it can be used by ESC4 on future runs.

5.2.1 Example of OUA-ESC

As an example, consider the method in Figure 34, which computes the integer square root Newton's method[5]. The ATPs used by ESC4 are able to discharge 14 of the 19 sub-VCs generated for this method. For each of the remaining 5 sub-VCs, a separate `.thy` file is output. If a proof were given then ESC4 will be able to discharge the corresponding sub-VC during the next compilation cycle. The first of the 5 unproven sub-VCs corresponds to the loop invariant $x < (y + 1) * (y + 1)$ not holding on entry to the loop and is encoded as the main lemma shown in Figure 35. We next examine the contents of this file and how we created a proof of the sub-VC.

[5]This is an adaptation of an example given in the Why distribution [why, 2008].

```
public class IntSqrt {
   //@ requires x >= 0;
   //@ ensures  \result * \result <= x;
   //@ ensures  x < (\result + 1) * (\result + 1);
   public static int sqrt(int x) {
      if (x == 0)  return 0;
      if (x <= 3)  return 1;
      int y = x;
      int z = (x + 1) / 2;
      //@ maintaining z > 0 && y > 0;
      //@ maintaining z == (x / y + y) / 2;
      //@ maintaining x < (y + 1) * (y + 1);
      //@ maintaining x < (z + 1) * (z + 1);
      //@ decreasing y;
      while (z < y) {
            y = z;
            z = (x / z + z) / 2;
      }
      return y;
}   }
```

Figure 34: Calculating the integer square root

The theory file shown in Figure 35 starts by stating its name and importing the ESC4, the Isabelle/HOL-specific Universal Background Predicate (UBP), a collection of theorem and function definitions that can be used in proofs. Two lemmas follow: the second (main) is the one created by ESC4 that states the VC that could not be discharged. The first (helper) was added later, as described below. The proof originally left by ESC4 for main was the Isabelle keyword **oops**, which causes Isabelle to stop processing the lemma and ignore it. The VCs stored in the theory files are not very user-friendly, but having the ProofGeneral parse the lemma causes it to be pretty printed as the single subgoal to be discharged. This causes unnecessary typing information to be removed, and the structure of the expression is shown through proper indentation.

```
theory IntSqrt-sqrt-1
imports ESC4
begin

lemma helper: (0::int) < x ==> x < (x + 1) * (x + 1)
 by (metis add1-zle-eq eq-iff-diff-eq-0 int-one-le-iff-zero-less
     linorder-not-less mult-less-cancel-left2 order-le-less-trans
     order-less-le-trans pordered-ring-class.ring-simps(27) ring-class.ring-simps(9)
     zadd-commute zle-add1-eq-le zle-linear zless-add1-eq zless-le)

lemma main: ([| (True & ((x-162-0::int) >= (0 :: int))); (~ ((x-162-0::int) = (0 ::
int))); (~ ((x-162-0::int) <= (3 :: int))); ((y-227-0::int) = (x-162-0::int)); ((z-240-0::int)
= (((((x-162-0::int)) + ((1 :: int)))) div ((2 :: int))))|] ==>  ((x-162-0::int) <
(((((y-227-0::int)) + ((1 :: int)))) * ((((y-227-0::int)) + ((1 :: int)))))))
 apply (auto)
 apply (simp add: helper)
done

end
```

Figure 35: A proof for a VC from the code in Figure 34

To prove this lemma, we opened its file in ProofGeneral, removed the **oops**, and started our proof by applying auto. The sub-VC's original lemma is not very user friendly, but auto is usually able to simplify it greatly. In this case, the single subgoal left is simply

$$\neg x \leq 3 \Longrightarrow x < (x + 1) * (x + 1).$$

We copied and pasted this subgoal as a separate helper lemma above the main lemma. If we could prove this helper then Isabelle would be able to prove the main lemma using its simplification procedures.

As a general rule, we first try to find a proof for helper lemmas using Isabelle's **sledgehammer** command [Paulson and Susanto, 2007] to find a Metis proof. Metis [met, 2008] is an ATP that tries to prove lemmas using a given list of of theorems. Isabelle's **sledgehammer** command tries to find such a list automatically.

When this command is able to find a proof, it can be copied and pasted directly into the proof script.

Unfortunately in our case, **sledgehammer** failed to find a proof. We noticed that the helper lemma could be generalized by changing the implication's antecedent from

$$\neg x \leq 3$$

to

$$x > 0,$$

which was enough to allow **sledgehammer** to find a Metis proof. Once the helper lemma was proved, we were able to prove the main lemma by adding the helper to the facts used by the simplificaiton procedure.

If **sledgehammer** is unable to find a proof for a useful helper lemma, it may still be provable manually by crafting an explicit proof. The Isar language [Wenzel, 1999] provides a structured way of expressing proofs in a form that is similar to pen-and-paper proofs but that can still be checked by Isabelle. We comment more about the relevance of Isar in the next subsection.

5.2.2 Discharging Helper Lemmas

Being able to provide both Metis and Isar proofs fits nicely into an iterative development cycle. Initially, developers may only be interested in knowing that the lemma holds true, and a slegehammer-provided Metis proof is sufficient since there is no concern for the readability or maintainability of the proof. This is especially true during active development, as the lemmas needed for a method may change frequently. Also, Metis proofs may be unstable over time, as the heuristics that the Metis prover uses to find a proof from a list of theorems could be subject

to change. As the system becomes more stable, it may be of interest to develop an explicit Isar-style proof as this may give insights into the problem domain.

In our example, a Metis proof was found for the helper that we could copy and paste into the theory file. Executing the proof of this particular helper lemma takes a long time, but once it and the original lemma are proved inside ESC4, the sub-VC is stored in the method's VC cache and Isabelle is not asked to prove it again.

5.2.3 Summary

Offline User-Assisted ESC provides users who are willing to put in the effort of developing proofs with a way to verify much more code than classical ESC, which relies solely on ATPs, can. This benefit is provided without the burden of forcing users who are not willing (or able) to generate the needed proofs with doing anything extra. The end result is an overall verification technique that offers a level of completeness in verification that is proportional to the effort the end user is willing to invest (and this is usually proportional to the criticality of the code).

Chapter 6

Distributed and Multithreaded Verification[1]

> Divide ut imperes. (*Divide so that you may rule.*) — Roman maxim
>
> Many hands make light work. — English proverb

Applying ESC to industrial-scale applications has been difficult because of the time existing tools require to generate and discharge VCs. In this chapter we highlight the enhancements that have been added to ESC4 that reduce the time needed to verify JML-annotated Java code.

While small classes can be verified in seconds with previous ESC tools, larger programs of 50 KLOC can sometimes take hours to verify. We believe that this is an impediment to widespread adoption of ESC: Being used to modern incremental software development models, developers have come to expect that the compilation (and ESC) cycles are very quick.

- We take advantage of the inherent modularity of the verification techniques underlying ESC [Leino, 1995] to analyze the methods in a given compilation

[1] This chapter is based on [James *et al.*, 2008].

unit in parallel. This is possible because the ESC analysis done for a given method is independent of that for any others. (Section 6.1)

- We take advantage of ESC4's proof strategies to develop distributed discharging so that non-local resources can be used to reduce the time to verify a set of classes. (Section 6.2)

- The previous two points are achieved by means of OS-independent proof services: If an executable version of a given prover is not available for a given platform, that prover can be exposed through a service and used remotely as if it were local. (Section 6.3)

Tools exist for verifying distributed and multi-threaded code, but we have not found another verifier that makes use of these techniques to speed up its own analysis. We believe that ESC4 is the first fully automatic static-verification tool to do so.

6.1 Multi-threading

The biggest gains in speed in ESC4 come from making use of all available CPU resources. Using the arguments in Leino's thesis, *Toward Reliable Modular Programs* [1995], it can be shown that each JML-annotated method in a system can be verified independently of the others. Where there are no dependencies, it is possible to introduce concurrency.

First-generation tools such as ESC/Java [Flanagan *et al.*, 2002] and ESC/Java2 [Cok and Kiniry, 2005] were written before multi-threaded and multi-core computers were commonplace. Multi-threading operating systems were already available then, but writing the code to use them would have only increased its

complexity without making the processing any faster. This encouraged a serialized approach to the problem, even though the modular nature of ESC is inherently parallelizable. Today, however, multiple-core machines are becoming the norm. Each thread could, in theory, run on its own core and thus reduce the time needed to verify a system to the most time needed to verify a single method. While the number of cores needed to achieve this level of speedup will not be available in the foreseeable future, having such small-grained units of work should make efficient scheduling easier for the operating system and/or virtual machine.

Modifying ESC4 to take advantage of ESC's inherent concurrency simply required adding a thread pool: Instead of processing each method sequentially, we packaged the processing (the body of an inner loop) as a work item and added it to the thread pool's task list. Finally, we added a join point to wait until all of the work for a compilation unit's methods finished before ending the ESC phase for it. This last step is necessary because the results of ESC may be used during code generation.

Version 3.4 of the Eclipse Java compiler added the ability to compile individual source files concurrently using multiple threads [mul, 2008]. Since ESC4 and JML4 are built on top of this compiler, all we had to do to gain this benefit was to ensure that JML4 is thread safe.

The vast majority of the time doing ESC is spent discharging VCs. Specifically, it is the underlying theorem provers that use the most time. For this reason, most ESC tools only make use of a single ATP per verification session. As mentioned above, ESC4 uses three by default, and 2D VC Cascading can cause those three to be invoked multiple times for each method. Just as the methods in a class can be verified in parallel, the sub-VCs for a method can be discharged in parallel. We

just need to put a join point so that we know when the processing of a method's VC has finished.

This gives ESC4 3 layers of parallelism: source files, methods within those files, and sub-VCs for those methods.

6.2 Distributed VC Processing

Once we were able to take advantage of all of the CPU resources on a local machine, it became interesting to ask if we could make use of resources on remote machines. The design of ESC4's Prover Coordinator led to quick discovery of a few deployment scenarios for the distributed discharging of VCs. It was easy to support distributed provers by adding new strategy communication infrastructure.

1. **Prove whole VC remotely.** The first deployment scenario offloads the work of the Prover Coordinator for an entire method. This was done by developing a new subclass of IProverStrategy that sends the VC generated for a method to a remote server for processing. (see Fig. 6.2). A Prover Coordinator is instantiated on the remote server along with its strategies. We initially had it behave like a local Prover Coordinator and discharge the VC itself with its own local provers.

2. **Prove sub-VCs remotely.** A second deployment scenario was to split the VC into sub-VCs and send each of them off for remote discharging. This was done by extending the ProveVcPiecewise strategy discussed in Section 4.2.1 and having it use remote services to discharge the sub-VCs in parallel.

3. **Doubly Remote Prover Coordinator.** Combining these two approaches, so that the remote Prover Coordinator itself delegates the responsibility for

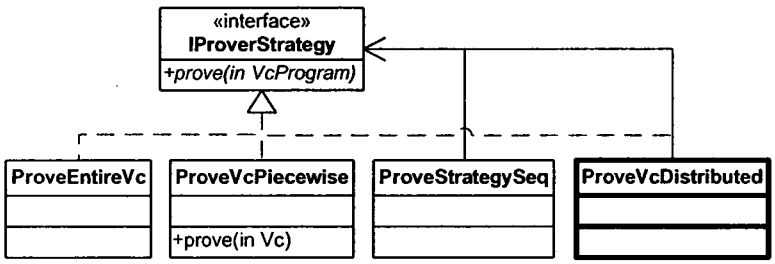

Figure 36: ESC4's distributed prover back-end

discharging the sub-VCs to remote services by using the ProveVcPiecewise-Distributed strategy, provides yet another alternative. A deployment view can be seen in Fig. 37.

Scenario 1 uses the least bandwidth, since only the original VC is transmitted. Scenario 2 uses the next least, although it can be exponentially more than 1. Scenario 3 uses the most, the sum of 1 and 2, but it is split into two groups: the same is used between the local machine and the remote Prover Coordinator as in 1, and between the remote Prover Coordinator and its servers as in 2.

Splitting a VC into sub-VCs can cause exponential growth in size, since these sub-VCs each represent a single acyclic path from the method's precondition, through its implementation to an assertion. As a result, scenarios 1 and 3 would be preferred over 2 when the remote machines are not on the same local area network. Scenario 3 can be thought of as providing the best parts of the other two: low bandwidth requirements to reach the prover service, and 2D VC Cascading. In addition, scenario 3 is the most likely to be used when a large farm of servers is available or when the Prover Coordinator service provides a façade that hides load balancing and other details from ESC4.

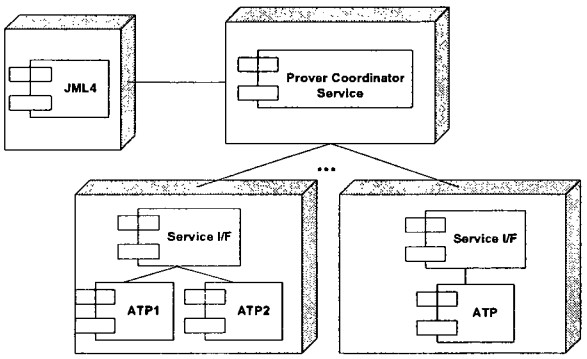

Figure 37: Deployment

6.3 Prover service

Independent of the strategy used, the proving resources may be local or remote. The initial prover adapters communicated with local resources using Java's Process mechanism. After facing some difficulties installing some provers on all of our development platforms, we hit on the idea of Prover Services.

The adapters that use the provers locally can be taken as base classes to subclasses that access them remotely. Part of the purpose of the adapter classes is to hide the interface with the provers. Applying the same concept lets us hide whether the prover is hosted locally or on a remote machine.

This has the advantage of making the provers OS independent. If a prover is needed on an OS for which there is no executable, it can be hosted on another machine with the appropriate OS and an adapter can hide the extra communication needed to access it.

6.4 Validation

To confirm that our approach produces speedups, we performed some preliminary timing tests. The source tested was a single Java class with 51 methods. For this code, ESC4 produced 235 VCs. Table 2 shows the number of times each of provers was invoked. Simplify was able to discharge over 80% of the VCs. It was also able to show as false almost 80% of those that were indeed false $(23 + 6)$. In this sample, CVC3 was not able to prove any of the VCs that Simplify was also unable to prove, and Isabelle was needed for just over 5% of the original VCs.

We ran the test with two deployment scenarios, both based on the Doubly Remote Prover Coordinator described in Section 6.2. In the first, the Prover Coordinator was hosted on the same PC as ESC4. In the second, it was hosted on a faster remote machine.

ESC4 was run on a 2.4 GHz Pentium 4. The Prover Coordinator was hosted either locally to the ESC4 machine or on a 3.0 GHz Pentium 4. Neither of these machines' CPUs is hyperthreaded. The provers were hosted on servers, each with a 2.4 GHz Quad-core Xeon processor. The timing results are shown in Table 3. Each entry in the last two columns is the average of three test runs, which were made after an initial run with the configuration being tested to remove initialization costs. Even so, the timings varied from 0.5 s to 1.6 s. Network usage may account for some of this variation.

For comparison, running the test with the Prover Coordinator and provers were all on the same PC as ESC4 took 72 s. It should be noted that when using remote provers, the CPU of the local machine stayed at 100% during the first few seconds and then dropped to below 20% while gathering the results. When the Prover Coordinator was on a separate machine, that machine's CPU was never went above 50%.

Table 2: VCs discharged with provers

Prover	No. VCs	No. Proved	(%)
Simplify	235	193	82
CVC3	42	0	0
Negation[a]	42	23	55[b]
Isabelle	19	13	68
failed	6		

[a] Simplify used to prove the negation of the VC

[b] 80% of all false

Table 3: Timing results

No. servers	No. cores	Time (s) with Prover Coordinator local	remote
1	4	26.6	26.4
2	8	16.9	16.2
3	12	12.8	13.3

The data gathered indicate that there is little difference between hosting the Prover Coordinator locally or remotely. We had thought that hosting it remotely would allow the VCs to reach the provers faster, thus giving a greater speedup. Surprisingly, as more processing cores were made available, it was actually faster to send the VCs directly. Further testing will have to be done to confirm this. For the sample shown, the timing difference between the two scenarios is within the range of error.

A function from the number of processors used to the time taken to analyze a given piece of code can be derived by applying simple algebra to Amdahl's law [Amdahl, 1967, Krishnaprasad, 2001]. It should have the form

$$t = C_1 + \frac{C_2}{n},$$

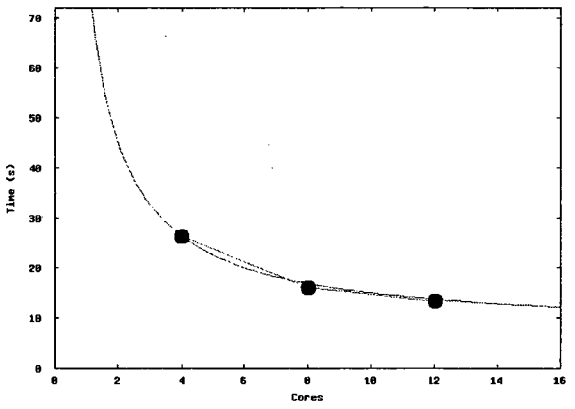

Figure 38: Time (s) vs. Cores

where C_1 is the time taken to complete the portion that cannot be serialized and C_2 is the time for the portion that can. Replacing n with 4 and 8 cores and t with the times for the remote Prover Coordinators gives a system of 2 linear equation with 2 unknowns. Solving this system gives

$$t = 7.4 + \frac{76.0}{n}$$

The experimental result of 13.3 s for 12 cores is within the error range of the predicted time of 13.7 s. The curve and data are shown in Figure 38.

These initial results with up to 12 cores suggest that over 90% of the ESC analysis is amenable to parallelization. One question that future study will have to address is, "Can the 7.4 s that was not parallelized by using distributed provers be made parallelizable by hosting ESC4 on a multi-core machine?" Contained in the serial part is the JDT's front-end generation of the AST and ESC4's generation of VCs from it.

After adding 12 cores, the serial portion takes longer than the portion that is parallelized. We did not test the generation of VCs on a multi-core system.

Doing so may show that at least part, and maybe even most, of this segment is parallizable.

6.4.1 Other Tools

As noted in the introduction, we have not been able to find other existing tools that make use of distributed or parallel processing to enhance fully automatic program verification. Two related aspects of the work presented here have been previously examined: multi-threaded, distributed compilation and interactive, distributed theorem proving for program verification. These are discussed in the following subsections.

6.4.1.1 Compilation

As mentioned in Section 6.1, Eclipse 3.4 supports mutithreaded compilation of Java programs. The Gnu make command `gmake` has a `--jobs[==n]` option that executes up to *n* build tasks concurrently. If an integer *n* is not supplied then as many tasks are started as possible [gma, 2006]. Microsoft's Visual C++ compiler has the "Build with Multiple Processes" option (`/MP`) that launches multiple compiler processes. If no argument is given, the number of effective processors is used. The number of effective processors is the number of threads that can be executed simultaneously and considers the number of processors, cores per processor and any hyperthreading capabilities.

Several open-source projects and commercial products are available that can distribute the tasks in a build process to networked machines. These only launch a process on a remote machine and do not make use of a service-based approach. Open-source projects include distcc [dis, 2008] and Icecream [ice, 2006]. Xoreax

sells a product called IncrediBuild [inc, 2008] that coordinates distributed builds from within with Microsoft's VisualStudio.

6.4.1.2 Interactive, distributed theorem proving for program verification

Vandevoorde and Kapur describe the Distributed Larch Prover (DLP), "a distributed and parallel version of LP, an interactive prover" [Vandevoorde and Kapur, 1996]. Like LP, DLP is not an ATP, as users must guide the proof-discovery process. It achieves parallelism by allowing users to simultaneously try several techniques to prove a subgoal. This is done by distributing the attempts among computers on a network. Some automation is provided by heuristics that chose the inference methods to be launched in parallel.

Hunter et al. attempt to use distributed provers to increase the adoption of formal techniques in industry [Hunter *et al.*, 2005]. Like the DLP, their approach requires interaction, but their goal is to reduce that interaction. Reducing the amount of user interaction would reduce the cost of using formal tools to prove software correct and thus remove one of the impediments to its more widespread use. A user interacts with software agents that try to automatically prove a goal. User interaction is needed when one of these agents is unable to automatically prove subgoals.

6.5 Summary

Applying ESC to industrial-scale applications has been difficult because of the time required. Invoking a theorem prover for every method in a system is computationally expensive. We attacked this by applying the divide-and-conquer strategy to allow processing by multiple computing resources, both local and remote.

Generating and discharging the VC for Java methods is a problem that can be easily decomposed into many independent tasks. This makes it very amenable to multi-threading and distributed processing. Given the power of today's desktop PCs, most of an organization's desktop computers' CPUs are under-utilized. Installing a distributed proving service on these machines would allow the organization's developers to tap into existing resources without requiring the acquisition of additional hardware.

The Eclipse JDT compiler is able to process multiple source files in parallel. We showed how we modified ESC4 to support verifying multiple methods in parallel. Similarly, a method's sub-VCs are discharged in parallel. Because of the potential reduction in time to verify a system, it became useful to explore distributed prover resources. This in turn led to exposing individual provers as distributed resources. All of these combined make the verification of Java programs scalable: The time ESC4 needs to verify a system should be inversely proportional to the CPU resources made available to it.

Chapter 7

Language Enhancement: Monotonic Non-null[1]

> "The obscure we see eventually. The completely obvious, it seems, takes longer." — Edward R. Murrow

In this chapter we describe a follow-up to the analysis of the case study presented in Appendix E. One of our goals was to see if we could further reduce the use of nullable types. While our initial investigation allowed us to conclude that 75% of declarations of reference types are non-null, a natural follow-up question was: Can the remaining uses of nullable be categorized and if so, which more semantically meaningful categories exist aside from nullable?

In Section 7.1, we examine the uses of nullable references. We noticed that over half of the nullable fields in our sample follow what we call a *monotonic non-null pattern.* As a contribution, we define in Section 7.2 *monotonic non-null types* and describe the benefits of their use, particularly in the context of multithreaded programs. We also describe our enhancement to the JML4 IVE

[1]This chapter is based on [Chalin *et al.*, 2008c].

that supports this new concept through the introduction of the JML type modifier `eventually_non_null`.

7.1 Uses of Null

In the following we share our insights on the uses of nullable for fields, method return types, and method parameters that were found in our case study.

7.1.1 Fields

One of the first categories that we noticed, mainly because of its prevalence, was that some fields are not initialized during construction, but later on. A large group of these fields are only ever assigned non-null values. Once these fields are set to a non-null value, they remain non-null. This happens, for example, when fields are lazily initialized, as is the case for the `theUniqueInstance` field of the Singleton pattern [Gamma *et al.*, 1995]. We call this group *monotonic non-null* since their non-null status is monotonic.

Another use of nullable references is at the end of an object's life cycle. References to *large* objects (or objects that refer to a large number of other objects, such as a tree) are often declared nullable so that the reference can be set to null when the object is no longer needed. This is necessary in Java to reduce memory usage and avoid what is often referred to as *memory leaks* in Java [Bloch, 2001, Item 5].

Generally, the use of nullable fell into two groups: those that are monotonic non-null (i.e., those that are non-null once initialized, but that are not initialized until after object construction) and those that freely move between holding null

and non-null values during their lifetimes. The latter are similar to ML's `option` type [Paulson, 1991], which is defined as

```
datatype 'a option = None | Some 'a
```

It is interesting to note that the Nice language used the keyword `option` instead of nullable [Bonniot, 2005].

7.1.2 Methods

Methods return null either because it is a valid return value or to indicate that a valid object could not be returned for a number of reasons:

- initialization of an object has not been completed, hence some of its methods return null,
- an error occurred,
- no value corresponding to the parameters was found, e.g., in a database query or search of a data structure, or
- the end of a recursive data structure was reached, e.g., in a linked list.

All of these can be considered as cases of methods having an Option type. In the first two cases, methods could throw an exception instead of returning null. In other cases, the *Null Object* pattern [Fowler, 1999] could be applied.

7.1.3 Parameters

Method parameters were the only category of reference in which it was not possible to determine a meaningful refinement in the use of nullable types. That is, in all cases a null value either was a valid value for a setter method (e.g., for the

```
if (this.f != null) {
  this.f.g ... // possible NPE;
  // field can be changed by another thread
}
```

Figure 39: Testing of a field against null is useless in multithreaded programs

purpose of resetting a nullable field) or indicated a *don't care* or *not applicable* parameter. In the latter case, overloading the method to only require parameters of interest would eliminate the need for a nullable parameter.

7.1.4 Statistics

Most of the time, nullable references can be avoided using the Null Object pattern, Java's exception mechanism, or overloaded functions. Nonetheless, our empirical study shows that, on average, developers would use them for approximately one reference out of four. The most interesting usage of null that we identified was the monotonic non-null use for fields. Nullable fields are best avoided because it is not possible, in general, to reason statically about their nullity status in the context of multithreading. The problem is illustrated in Figure 39: while a nullable field can be tested for null in one thread, another thread can change that field's value between a test against null and a dereference. A Java programming idiom that allows one to safely test and then use the non-null value of a field is illustrated in Figure 40. This idiom is so common that a special syntax was introduced in Eiffel to address it; an Eiffel program fragment equivalent to the one of Figure 40 is given in Figure 41. Notice how the syntactic shorthand allows the local variable declaration to be embedded inside the `if`'s condition.

```
F f0;
if ((f0 = this.f) != null) {
  f0.g... // safe
}
```

Figure 40: An idiom for testing fields against null (that is thread safe)

```
if f0: F Current.f then
  f0.g... // safe
end
```

Figure 41: An *object test* in Eiffel

It is in the light of these examples that one can see the benefit of declaring fields as monotonic non-null; i.e., for monotonic non-null fields, the test in Figure 39 would be sufficient to guard against potential null dereferences in the `then` clause of the `if` statement. In fact, from the point of view of flow analysis, monotonic non-null fields can be treated like method parameters, for which the nullity status can always be determined. In reviewing our case-study subjects a second time, we noted that approximately 60% of nullable fields were monotonic non-null. Details, per project of our study, are shown in Table 4.

The concept of *monotonic non-null* applies not only to fields but also to method return values to a certain extent. An obvious group of methods that return a monotonic non-null value is the getter methods for monotonic non-null fields.

Table 4: Proportion of nullable fields that are monotonic non-null

	JML Checker	ESC/ Java2	SoenEA	Koa TS	Eclipse JDT Core	Sum or Average
Mono non-null	11	14	9	9	15	58
Option type	3	13	4	12	7	39
Total	14	27	13	21	22	97
% Mono non-null	78.6%	51.2%	69.2%	42.9%	68.2%	59.8%

This is just one case of pure methods that may initially return null but after sufficient initialization always return a non-null value.

7.2 Monotonic Non-null

In the previous section, we introduced a group of nullable references we call *monotonic non-null.* This concept has been implemented, through the use of the `eventually_non_null` keyword in JML4 [Chalin *et al.*, 2007] [Chalin *et al.*, 2008a]. Field declarations annotated with this modifier are not guaranteed to be initialized to a non-null value by their declaring class's constructors, but like `non_null` fields, they are not allowed to be assigned a nullable value. Monotonic non-null references behave like non-null references once they have been assigned to. Because of this, we are able to treat them as non-null after a simple test against `null`.

While monotonic non-null fields share some similarities with non-null fields of a Raw or existentially Delayed object, as described in [Fähndrich and Leino, 2003] and [Fähndrich and Xia, 2007], they are a more general and flexible concept. Like our monotonic non-null fields, non-null fields of an instance of a Raw or Delayed type can be null until they are assigned to but, in contrast, an instance of a raw or delayed type must become *cooked* before the end of the constructor's body is reached. An instance is cooked if *all* of its non-null fields have been assigned *non-null* values. Hence, a main difference is that Raw or Delayed is a type qualifier that applies to a class type and it influences the semantics of all non-null member fields of that type. Monotonic non-null is a per-field qualifier. Thus, monotonic non-null fields can remain null beyond the end of an object's constructor's body, and if the object has more than one monotonic non-null field then the nullity status of each can change independently. While in the general setting of multithreaded programs this construct can help to provide safety from `NullPointerExceptions`, it does not eliminate the possibility of race conditions.

```
/*#@ eventually_non_null */ Object f;

/*@ ensures \result == this.f; */
/*@ pure */ /*#@ eventually_non_null */ Object m() {
        return this.f;
}
```

Figure 42: A monotonic non-null method

```
/*@ nullable */ Object f;
//@ constraint \old(f) != null ==> f != null;
//@ constraint \old(m()) != null ==> m() != null;
//@ ensures \result == this.f;
/*@ pure nullable */ Object m(){
        return this.f ;
}
```

Figure 43: Same code desugared to standard JML

In addition to field declarations, methods return types can also be annotated as `eventually_non_null`. Once such a method returns a non-null value, it is guaranteed to never again return `null`. As we mentioned earlier, getter methods for `eventually_non_null` fields are obvious candidates. Figure 42 shows such a method[2]. Monotonic non-null methods can be desugared using, in particular, JML's constraint clause as shown in Figure 43. A constraint clause, also called a history constraint, expresses properties that must hold between any visible state (whose values are captured via the `\old()` operator) and all visible states that follow it [Leavens *et al.*, 2008, §8.3]. Since `eventually_non_null` is a type modifier, desugaring it into a constraint on the field is not strong enough, i.e., it is only an approximation of its true meaning.

[2]The # before the @ in the first line indicates JML4-specific annotation.

Parameters are better behaved than fields in that their initial values are fixed at the point of call and flow analysis can track their nullity status within the method body. As a result, monotonic non-null is not a useful modifier for formal parameters. The type parameters of generic types can also take nullity attributes, but we have not come across a case in which it would be useful to have these be `eventually_non_null` instead of `nullable`.

Arrays whose elements are marked as `non_null` but whose declaration does not have an initializer are almost always meant to have `eventually_non_null` elements. This is one possible solution to the problem of determining an ending point for their initialization [Fähndrich and Leino, 2003].

We have added compile-time and runtime checking of `eventually_non_null` fields to the JML4 compiler [Chalin *et al.*, 2007]. Static checking is accomplished by simply disallowing an assignment of a value *not known to be non-null*[3] to such a field. At runtime, a contract-violation error is thrown when the right-hand side of an assignment to a field declared to be `eventually_non_null` evaluates to `null`. Checking that the value returned by a method is `eventually_non_null` is beyond the abilities of the type system, and would have to be performed, e.g., by extended static checking using the desugaring that was given earlier. Runtime checking would require keeping an extra Boolean field, initialized to `false`, that to indicate whether the method has returned non-null. When an `eventually_non_null` method terminates, if the value to be returned is `null` but the Boolean field indicates that a non-null value has already been returned then a contract violation error should be thrown. If the value to be returned is non-null then the Boolean field is set to `true`.

[3]Note that this is a distinct category from values known to be null.

7.3 Summary

Based on an analysis of the usage of nullable types, we discovered the prevalence of monotonic non-null: Almost 60% of the nullable fields in our study were of this type. We demonstrated how the use of monotonic non-null types could be beneficial, particularly in the context of multithreaded programs. Monotonic non-null types have been partially implemented in JML4 and are available through the use of the `eventually_non_null` type modifier.

Chapter 8

Conclusions

> "Chaque chose que nous voyons en cache une autre, nous désirons toujours voir ce qui est caché par ce que nous voyons."
> *(Everything we see hides another. We always want to see that which is hidden by what we see.)*
> — René Magritte

8.1 Summary

The work presented in this thesis falls in five main areas, each with subprojects.

1. We explored a Non-Null Type System (NNTS) for JML.
   - After quantifying relative usages of nullable and non-null references,
   - we analyzed the uses of nullable references, and
   - we introduced syntax and semantics for new nullity modifier for declarations of reference types.

- We implemented a NNTS within Eclipse's Java compiler, including full support for the monotonic non-null modifier. This support was later extended to RAC and ESC.

2. We laid the foundations of JML4, an Eclipse-based IVE for JML.
   - It became the basis for the JML Community's second generation of tools.
   - We led a JML Winter School to train other researchers to work on JML4.
3. We developed ESC4, a from-scratch rewrite of (a subset of) ESC functionality for JML4. Notable features include
   - improved coverage by allowing verification of some constructs similar tools cannot
   - improved completeness with 2D VC Cascading and
   - improved usability by indicating provably false assertions.
4. We sped up the processing of ESC4
   - by using multiple threads to generate VCs in parallel and
   - by using non-local prover resources to distribute VC discharging.
   - Preliminary validation was reported.
   - Proof-status caching was used to reduce the time to reverify code.
     - We proposed ways of reducing the fragility of the cache by removing source-position information from the items stored cache.
5. A novel form of ESC was introduced: Offline User-Assisted ESC, which
   - improves completeness by allowing users to take advantage of the full power of Isabelle and

- helps when debugging code and specifications by isolating unprovable propositions.

Even after Java falls from common use, many of these contributions will remain vaild.

8.2 Future Work

> "To explain all nature is too difficult a task for any one man or even for any one age." — Isaac Newton

8.2.1 Preparing ESC/Java2 for the VSR

Continuing to run RAC-instrumented versions of ESC/Java2 will uncover further bugs and design issues. ESC/Java2 can be used to analyze the source for the JML Compiler, as it also has a large JML-annotated code base. Using the tools iteratively to analyze both themselves and each other should enhance their quality, hence making them more likely potential candidates for inclusion in the Verified Software Repository.

8.2.2 JML4

There is still much work to be done on JML4. The parser recently reached support for JML language-level 2, but the type-resolution and static-analysis phases have yet to reach level 0. Once the compiler front end is done, it will be necessary to propagate support for JML constructs into the RAC, ESC, and FSPV components. Fairly pressing is the handling of JML4's math modes, which should be reexamined and fully implemented.

We had thought that completion of the front end would also mark a milestone after which developers of other JML tools would be able to explore the possibility of integrating their tools within the JML4 framework, but as mentioned in Section 3.3.1.1, some groups are already using JML4 as their front end. Other interested researchers should be encouraged to contribute to this effort and incorporate their verification components.

JML4-based second-generation versions of existing tools, such as JmlUnit (see Section 2.2.3), could be developed. JML4 could be extended to support more advanced IDE functionality such as specification refactoring, browsing, folding, and navigation.

8.2.3 ESC4

ESC/Java2 can verify many constructs that ESC4 cannot. To close this gap, ESC4 should more fully support Java and JML. Full support for fields and arrays is currently missing from ESC4, and these will be needed before any substantial amount of code can be analyzed. ESC4 currently treats Java's integral types as unlimited precision, but once JML's math modes are supported, this should be corrected. Since Simplify does not support limited integral types, another ATP would have to be found that could provide information about VCs that cannot be discharged.

The interfaces to the theorem provers are very inefficient and could be made much quicker. Also, other more powerful ATPs, such as Z3, could be supported.

8.2.4 OUA-ESC

The VCs stored in the Isabelle theory files are not very user-friendly, and future work is unlikely to make them more palatable. Simply having Isabelle parse the

lemma causes it to be pretty printed as the single subgoal to be discharged. This causes unnecessary typing information to be removed, and the structure of the expression is shown through proper indentation.

8.2.5 Distributed Discharging of VCs

We modified ESC4 to take advantage of many local and non-local computing resources. The implementation was done to quickly get a usable and stable framework in place, without much regard for optimization. While we are pleased with the initial results, there are ample opportunities for improvement. These include using more efficient communication mechanisms to interact with remote resources. Load balancing and other techniques from service-oriented architectures are obvious candidates for consideration. Some of these are being investigated by a SOEN 490 Capstone group.

After making the obvious enhancements, timing studies could be conducted to evaluate the deployment scenarios mentioned in this paper, varying the number and kinds of local and remote resources as well as the characteristics (speed and reliability) of the network.

Bibliography

[Abrial, 1996] J.-R. Abrial. *The B-book: Assigning programs to meanings.* Cambridge University Press, New York, NY, 1996.

[Ahrendt *et al.*, 2005] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Amdahl, 1967] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, pages 79–81, San Francisco, CA, 1967.

[ban, 2009] Papers. `http://bandera.projects.cis.ksu.edu/papers/index.shtml`, 2009.

[Barnes, 2006] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, Boston, MA, 2006.

[Barnett and Leino, 2005] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, 2005. ACM Press.

[Barnett *et al.*, 2004] M. Barnett, W. Naumann, and Q. Sun. 99.44% pure: Useful abstractions in specifications, 2004.

[Barnett *et al.*, 2005] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop, Marseille, France, March 10-14, 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[Barnett *et al.*, 2006] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*

2005, Revised Lectures, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2006.

[Barthe *et al.*, 2007] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: A tool for validation of security and behaviour of Java applications. In *FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[Bicarregui *et al.*, 2006] J. C. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: A step towards the verifying compiler. *Formal Aspects of Computing*, 18(2):143–151, 2006.

[Bjørner and Jones, 1978] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[Bloch, 2001] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[Böhme *et al.*, 2008] Sascha Böhme, Rustan Leino, and Burkhart Wolff. HOL-Boogie – An interactive prover for the Boogie program verifier. In *Proceedings of the 21th International Conference on Theorem proving in Higher-Order Logics (TPHOLs 2008)*, LNCS 5170. Springer, 2008. Also available as `http://www-wjp.cs.uni-sb.de/publikationen/boehme˙tphols˙2008.pdf`.

[Bonniot, 2005] Daniel Bonniot. The Nice Programming Language. `http://nice.sourceforge.net`, 2005.

[bui, 2008] Building bug-free O-O software: An introduction to Design by Contract. `http://archive.eiffel.com/doc/manuals/technology/contract/`, 2008.

[Burdy and Requet, 2002] Lilian Burdy and Antoine Requet. JACK: Java applet correctness kit. In *4th Gemplus Developer Conference*, November 12-14 2002.

[Burdy *et al.*, 2003] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In *Proceedings of the International Symposium of Formal Methods Europe (FME'03)*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439, 2003.

[Burdy *et al.*, 2005a] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[Burdy *et al.*, 2005b] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[Burdy *et al.*, 2007] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering (FASE 2007)*, volume 4422 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 2007.

[c4j, 2007] C4J: DBC for Java. Design by Contract for Java made easy. `http://c4j.sourceforge.net`, 2007.

[Campbell-Kelly, 1989] Martin Campbell-Kelly, editor. *The early British computer conferences.* MIT Press, Cambridge, MA, 1989.

[Carter *et al.*, 2005] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software refinement with perfect developer. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 363–373, Washington, DC, 2005. IEEE Computer Society.

[Catano and Wahls, 2009] Nestor Catano and Tim Wahls. Executing JML specifications of Java Card applications: A case study. In *ACM SAC 2009 (24th Annual ACM Symposium on Applied Computing)*, 2009.

[Chalin and James, 2006] Patrice Chalin and Perry R. James. Cross-verification of JML tools: An ESC/Java2 case study. In *VSTTE '06: Proceedings of the 2006 workshop on Verified Systems: Theories, Tools, Experiments*, August 2006.

[Chalin and James, 2007] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP*, Berlin, Germany, 2007.

[Chalin *et al.*, 2006] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. Springer-Verlag, 2006.

[Chalin *et al.*, 2007] Patrice Chalin, Perry R. James, and George Karabotsos. An integrated verification environment for JML: Architecture and early results. In *SAVCBS '07: Proceedings of the 2007 Workshop on Specification and Verification of Component-Based Systems*, pages 47–53, 2007.

[Chalin *et al.*, 2008a] Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards an industrial grade IVE for Java and next generation research

platform for JML. In *VSTTE '08: Proceedings of the 2008 Conference on Verified Systems: Tools, Techiniques, and Experiments*, 2008.

[Chalin *et al.*, 2008b] Patrice Chalin, Perry R. James, and George Karabotsos. Using Isabelle/HOL for static program verification in JML4. In *Proceedings of TPHOLs: Emerging Trends*, pages 1–8, August 2008.

[Chalin *et al.*, 2008c] Patrice Chalin, Perry R. James, and Frédéric Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *IET Software Journal*, 2008.

[Chalin *et al.*, 2008d] Patrice Chalin, Perry R. James, Frédéric Rioux, and George Karabotsos. Towards a verified software repository candidate: Cross-verifying a verifier. Technical Report ENCS-CSE-DSRG-TR 2008-001b, Dependable Software Research Group, Concordia University, Montreal, Quebec, 2008.

[Chalin, 2005] Patrice Chalin. Logical foundations of program assertions: What do practitioners want? Technical Report 2005-002, Computer Science Department, Concordia University, June 2005. Also available as `http://www.cs.concordia.ca/$"sim$chalin/papers/TR-2005-002-r2.pdf`.

[Cheon and Leavens, 2001] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01–12, Department of Computer Science, Iowa State University, 2001.

[Cheon and Leavens, 2002] 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, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002. Also available as `ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf`.

[Clarke and Rosenblum, 2006] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3):25–37, 2006.

[Coffee, 2006] Peter Coffee. eweek labs review: Jtest8. `http://www.eweek.com/article2/0,1895,2032589,00.asp`, October 2006.

[Cok and Kiniry, 2005] David R. Cok and Joseph Roland Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362/2005 of *LNCS*, pages 108–128. Springer Berlin, 2005.

[Cok *et al.*, 2007] David R. Cok, E. Hubbers, and E. Rodríguez. Esc/Java2 Eclipse Plug-in. `http://sort.ucd.ie/projects/escjava-eclipse`, 2007.

[Cok, 2007] David R. Cok. Design Notes (Eclipse.txt). `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/trunk/docs/eclipse.txt`, 2007.

[Cok, 2008] David R. Cok. Adapting JML to generic types and Java 1.6. In *SAVCBS '08: Proceedings of the 2008 workshop on Specification and verification of component-based systems*, 2008.

[Con, 2008] Contract4j. `http://www.contract4j.org`, 2008.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[Dahl *et al.*, 1972] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured Programming.* Academinc Press, Inc., New York, NY, 1972.

[Deng *et al.*, 2007] Xianghua Deng, Robby, and John Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. Technical report, Kansas State University, 2007.

[Detlefs *et al.*, 1998] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq SRC, Palo Alto, CA, December 1998.

[Dhara and Leavens, 1996] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 258–267, Washington, DC, 1996. IEEE Computer Society.

[Dietl and Müller, 2005] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005. Also available as `http://www.jot.fm/issues/issue_2005_10/article1.pdf`.

[Dijkstra, 1976] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[dis, 2008] distcc: A fast, free distributed C/C++ compiler. `distcc.org`, 2008.

[Ernst and Coward,] M. Ernst and D. Coward. Annotations on Java Types. JCP.org, JSR 308. October 17, 2006.

[Ernst *et al.*, 2007] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[Fähndrich and Leino, 2003] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312, New York, NY, 2003. ACM Press.

[Fähndrich and Xia, 2007] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 337–350, New York, NY, 2007. ACM.

[Fähndrich *et al.*, 2006] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[Filliâtre and Marché, 2007] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. *Computer Aided Verification*, pages 173–177, 2007.

[Filliâtre *et al.*, 2008] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus verification tool for C programs: Tutorial and reference manual. `http://caduceus.lri.fr`, 2008.

[Filliâtre, 2003] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[Filliâtre, 2008] J.-C. Filliâtre. The WHY verification tool: Tutorial and reference manual. `http://why.lri.fr`, 2008.

[Flanagan and Leino, 2001] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag.

[Flanagan and Saxe, 2001] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 193–205, New York, NY, 2001. ACM Press.

[Flanagan *et al.*, 2002] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference*, pages 234–245, New York, NY, 2002. ACM Press.

[Fowler, 1999] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley Professional, Boston, MA, 1995.

[gma, 2006] Parallel - GNU 'make'. `http://www.gnu.org/software/automake/man"discretionary--"-"ual/make/Parallel.html`, 2006.

[Gosling *et al.*, 2005] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, Boston, MA, 3 edition, 2005.

[Hoare, 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoare, 2003a] C. A. R. Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[Hoare, 2003b] C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[Hunt *et al.*, 2005] Galen Hunt, James Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, WA, 2005.

[Hunter *et al.*, 2005] Chris Hunter, Peter Robinson, and Paul Strooper. Agent-based distributed software verification. In *ACSC '05: Proceedings of the Twenty-eighth Australasian Conference on Computer Science*, pages 159–164, Darlinghurst, Australia, 2005.

[ice, 2006] Icecream - openSUSE. `http://en.opensuse.org/Icecream`, 2006.

[iCo, 2001] iContract: Design by Contract in Java. `http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html`, 2001.

[ico, 2007] iContract2.org: Design by Contract for Java. `http://www.icontract2.org`, 2007.

[inc, 2008] IncrediBuild by Xoreax Software - Distributed Visual Studio Builds. `http://www.xoreax.com/"discretionary--"-"-"solutions vs.htm`, 2008.

[isa, 2008] Isabelle. `http://isabelle.in.tum.de`, 2008.

[Jacobs and Poll, 2003] B. P. F. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, September 2003.

[James and Chalin, 2009a] Perry R. James and Patrice Chalin. Enhanced extended static checking in JML4: Benefits of multiple-prover support. In *ACM SAC 2009 (24th Annual ACM Symposium on Applied Computing)*, 2009.

[James and Chalin, 2009b] Perry R. James and Patrice Chalin. Esc4: A modern caching ESC for Java. In *SAVCBS '09: Proceedings of the 2009 workshop on Specification and verification of component-based systems*, 2009.

[James and Chalin, 2009c] Perry R. James and Patrice Chalin. Faster and more complete extended static checking for the Java Modeling Language. *Journal of Automated Reasoning*, 2009. to appear.

[James *et al.*, 2008] Perry R. James, Patrice Chalin, Leveda Giannas, and George Karabotsos. Distributed, multi-threaded verification of Java programs. In *SAVCBS '08: Proceedings of the 2008 workshop on Specification and verification of component-based systems*, 2008.

[jas, 2007] Jass: Homepage. `http://csd.informatik.uni-oldenburg.de/~jass`, 2007.

[jCo, 2008] jContractor: Design by Contract for Java. `http://jcontractor.sourceforge.net`, 2008.

[Jézéquel *et al.*, 2001] Jean-Marc Jézéquel, Daniel Deveaux, and Yves Le Traon. Reliable objects: Lightweight testing for OO languages. *IEEE Softw.*, 18(4):76–83, 2001.

[Jones *et al.*, 2006] Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *Computer*, 39(4):93–95, 2006.

[Jones, 1990] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Upper Saddle River, NJ, 1990.

[jsr, 2008] JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308`, 2008.

[Karabotsos *et al.*, 2008] George Karabotsos, Patrice Chalin, and Leveda Giannas. Total correctness of recursive functions using JML4 FSPV. In *SAVCBS '08: Proceedings of the 2008 workshop on Specification and verification of component-based systems*, November 2008.

[Kiniry *et al.*, 2006] Joseph R. Kiniry, Alan E. Morkan, and Barry Denby. Soundness and completeness warnings in ESC/Java2. In *SAVCBS '06: Proceedings of*

the 2006 Workshop on Specification and Verification of Component-Based Systems, pages 19–24, New York, NY, 2006. ACM Press.

[Kiniry, 2007] Joseph Roland Kiniry. private communication, March 2007.

[Kolman and Busby, 1986] B Kolman and R C Busby. *Discrete mathematical structures for Computer Science (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1986.

[kra, 2009] Krakatoa. `http://krakatoa.lri.fr`, 2009.

[Krause and Wahls, 2006] Ben Krause and Tim Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, editor, *Formal Methods for Industrial Critical Systems (FMICS '06)*, volume 4346 of *Lecture Notes in Computer Science*, pages 293–296, New York, NY, 2006. Springer-Verlag. Also available as `http://users.dickinson.edu/~wahlst/papers/tool.pdf`.

[Krishnaprasad, 2001] S. Krishnaprasad. Uses and abuses of Amdahl's law. *The Journal of Computing in Small Colleges*, 17(2):288–293, 2001.

[Leavens and Cheon, 2005] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`, 2005. Draft, available from jmlspecs.org.

[Leavens *et al.*, 1998] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Java Modeling Language. In *Formal Underpinnings of Java Workshop*, 1998. (at OOPSLA'98).

[Leavens *et al.*, 1999] 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, 1999.

[Leavens *et al.*, 2000] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Department of Computer Science, Iowa State University, 2000.

[Leavens *et al.*, 2008] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph R. Kiniry, and Patrice Chalin. JML reference manual. `http://www.jmlspecs.org`, 2008.

[Leavens, 2009] Gary T. Leavens. The Java Modeling Language (JML). `http://www.jmlspecs.org`, 2009.

[Leino and Logozzo, 2005] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proceedings of the the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *Lecture Notes in Computer Science*, Tsukuba, Japan, November 2005. Springer-Verlag.

[Leino and Monahan, 2007] K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *FTfJP '07: Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs*, 2007.

[Leino *et al.*, 1998] K. Rustan M. Leino, Raymie Stata, James B. Saxe, and Cormac Flanagan. Java to guarded commands translation. Technical Report ESCJ 16c, Compaq, 1998. Available from the ESC/Java2 website.

[Leino *et al.*, 1999] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Report 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999.

[Leino, 1995] K. Rustan M. Leino. *Toward reliable modular programs.* PhD thesis, California Institute of Technology, Pasadena, CA, 1995.

[Leino, 2001] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 157–175, London, UK, 2001. Springer-Verlag.

[Liskov and Wing, 1994] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Language Systems*, 16(6):1811–1841, 1994.

[Liu *et al.*, 1998] Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Mitsuru Ohba, and Yong Sun. SOFL: A formal engineering methodology for industrial applications. *IEEE Trans. Softw. Eng.*, 24(1):24–45, 1998.

[Liu, 2004] Shaoying Liu. *Formal Engineering for Industrial Software Development: Using the Sofl Method.* Springer-Verlag, Berlin, 2004.

[met, 2008] Metis theorem prover. `http://www.gilith.com/software/metis`, 2008.

[Mey, 2005] EiffelWorld Column by Dr. Bertrand Meyer. `http://www.eiffel.com/general/monthly column/2005/April.html`, 2005.

[Mey, 2007] EiffelWorld Column by Dr. Bertrand Meyer. `http://www.eiffel.com/general/monthly column/2007/01.html`, 2007.

[Meyer *et al.*, 2000] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. `http://www.informatik.fernuni-hagen.de/pi5/publications.html`, 2000.

[Meyer, 1995] Bertrand Meyer. *Object success: Aa manager's guide to object orientation, its impact on the corporation, and its use for reengineering the software process.* Prentice-Hall, Inc., Upper Saddle River, NJ, 1995.

[Meyer, 1997] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, second edition, 1997.

[Meyer, 2005] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In Andrew P. Black, editor, *ECOOP 2005 - Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, UK,* volume 3586 of *Lecture Notes in Computer Science,* pages 1–32. Springer, 2005.

[Mitchell *et al.*, 2002] Richard Mitchell, Jim McKim, and Bertrand Meyer. *Design by contract, by example.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 2002.

[mob, 2009] Mobius: Webhome. `http://mobius.inria.fr`, 2009.

[mod, 2007] Nabble: JML5 proposal and Modern Jass. `http://www.nabble.com/JML5-Proposal-and-Modern-Jass-t3156880.html`, 2007.

[mul, 2008] Bug 142126 - utilizing multiple CPUs for Java compiler. `https://bugs.eclipse.org/bugs/show_bug."discretionary--"-"-"cgi?id=142126`, 2008.

[Nipkow *et al.*, 2000] Tobias Nipkow, David Von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In *Foundations of Secure Computation. Volume 175 of NATO Science Series F: Computer and Systems Sciences., IOS,* pages 117–144. IOS Press, 2000.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic,* volume 2283 of *LNCS.* Springer, 2002.

[Nipkow *et al.*, 2005] Tobias Nipkow, David von Oheimb, Cornelia Pusch, and Gerwin Klein. Project bali. `http://isabelle.in.tum.de/Bali`, 2005.

[ova, 2007] `http://oval.sourceforge.net`, 2007.

[par, 2009] Parasoft homepage. `http://www.parasoft.com`, 2009.

[Park, 1992] Robert E. Park. Software size measurement: A framework for counting source statements. Technical Report CMU/SEI-92-TR-20, CMU, Software Engineering Institute, Pittsburgh, PA, October 1992.

[Paulson and Susanto, 2007] Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2007*, LNCS 4732, pages 232–245. Springer, 2007. Also available as `http://www.cl.cam.ac.uk/~lp15/papers/Automation/reconstruction.pdf`.

[Paulson, 1991] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Potter *et al.*, 1996] Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z.* Prentice-Hall, Upper Saddle River, NJ, 1996.

[Rioux, 2006] Frédéric Rioux. Effective and efficient design by contract for Java. Master's thesis, College of Engineering and Computer Science, Concordia University, Montreal, Quebec, 2006.

[Robby *et al.*, 2008] Robby, Patrice Chalin, David R. Cok, and Gary T. Leavens. An evaluation of the Eclipse Java Development Tools (JDT) as a foundational basis for JML reloaded. `jmlspecs.svn:/reloaded/planning`, 2008.

[Rushby, 1993] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.

[Sarcar, 2009] Amritam Sarcar. Runtime assertion checking support for JML on Eclipse platform. In *CAHSI 2009 (3rd. Annual Meeting for Computer Alliance for Hispanic Serving Institutions)*, pages 79–82, 2009.

[Schumann, 2001] Johann M. Schumann. *Automated Theorem Proving in Software Engineering.* Springer-Verlag, New York, NY, 2001.

[sim, 2000] Simplify theorem-prover. `http://research.compaq.com/SRC/esc/Simplify.html`, 2000.

[spe, 2007] Spec# home. `http://research.microsoft.com/specsharp`, 2007.

[stc, 2007] STclass: A contract based built-in testing framework (CBBT) for Java. `http://www-valoria.univ-ubs.fr/stclass`, 2007.

[Turing, 1949] Alan M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation*, pages 67–69, Cambridge, UK, June 1949. University Mathematical Laboratory, Cambridge University. Reprinted in [Campbell-Kelly, 1989, 70–72]. Also available as `http://www.turingarchive.org/browse.php/B/8`.

[(UKCRC), 2006] UK Computing Research Committee (UKCRC). Grand challenges for computer research, 2006.

[van den Berg and Jacobs, 2001] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[Vandevoorde and Kapur, 1996] Mark T. Vandevoorde and Deepak Kapur. Distributed Larch Prover (DLP): An experiment in parallelizing a rewrite-rule based prover. In *RTA '96: Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, pages 420–423, London, UK, 1996. Springer-Verlag.

[Ver, 2007] Project-lemme:verificard. `http://ralyx.inria.fr/2003/Raweb/lemme/uid72.html`, 2007.

[vsi, 2008] The verified software initiative. `http://qpq.csl.sri.com/vsr/vsi.pdf/view`, 2008.

[Wenzel, 1999] Markus Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 167–184, London, UK, 1999. Springer-Verlag.

[why, 2008] Why: software verification platform. `http://why.lri.fr`, 2008.

[Wilson *et al.*, 2005] Thomas Wilson, Savi Maharaj, and Robert G. Clark. Omnibus: A clean language and supporting tool for integrating different assertion-based verification techniques. In *Proceedings of REFT 2005*, Newcastle, UK, July 2005.

[Wilson *et al.*, 2006] Thomas Wilson, Savi Maharaj, and Robert G. Clark. Push-button tools for application developers, full formal verification for component vendors. Technical report, Department of Computing Science and Mathematics, University of Stirling, Stirling, Scotland, December 2006.

[Wilson *et al.*, September 2005] Thomas Wilson, Savi Maharaj, and Robert G. Clark. Omnibus verification policies: A flexible, configurable approach to assertion-based software verification. In *SEFM'05, The 3rd IEEE International Conference on Software Engineering and Formal Methods*, September 2005.

[win, 2009] WineHQ - Run Windows applications on Linux, BSD, Solaris and Mac OS X. `http://www.winehq.org`, 2009.

[Winskel, 1993] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, 1993.

[Witte, 2003] M. Witte. Portierung, Erweiterung und Integration des Object-Teams/Java Compilers für die Entwicklungsumgebung Eclipse, 2003. Technische Universität Berlin.

[Woodcock, 2006] J. C. P. Woodcock. Grand Challenge 6: Dependable Systems Evolution, 2006.

Appendix A

Soundness and Completeness Proof for VC-Splitting Algorithm[1]

theory *Vc2Vcs* **imports** *Main* **begin**

A.1 Introduction

ESC4 produces a single VC for each method to be verified. We would like to split unprovable VCs into a collection of sub-VCs, the conjunction of which is equivalent to the original. This Appendix describes the decomposition algorithm as well as provides a proof that it is sound and complete.

A.2 VC Language and Splitting Algorithm

typedecl *BExp*

datatype *VC* =
VcIf VC VC
| *VcAnd VC VC*
| *VcAndAnd VC VC*
| *VcOther BExp*

The VC language that we work with in this Appendix contains only conjunction, implication, and undefined Boolean expressions. Two forms of conjunction are supported: logical and conditional. The former always evaluates both

[1]This Isabelle/HOL 2009 proof [James and Chalin, 2009c] is available online at `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/jml4/trunk/org.eclipse.jdt.core/notes/esc4/Vc2Vcs.thy`

operands, while the latter only evaluates its second operand if the first evaluates to *True*.

fun *distribImp* :: *VC* ⇒ *VC list* **where**
  distribImp (*VcIf a b*) = *map* (*VcIf a*) (*distribImp b*)
| *distribImp* (*VcAnd a b*) = *distribImp a* @ *distribImp b*
| *distribImp* (*VcAndAnd a b*) = *distribImp a* @ *map* (*VcIf a*) (*distribImp b*)
| *distribImp* (*VcOther b*) = [*VcOther b*]

The VC splitting is accomplished by distributing implications over the conjunctions, while keeping the conditional definition of the conditional conjunction. This special treatment was needed to provide proper error reporting.

A.3 Semantics

consts *M'* :: *BExp* ⇒ *bool*

To be able to show that our approach is sound and complete, we must first give meaning to VCs. Since *VcOther* is meant to be an uninterpreted boolean expression, its meaning is given by the uninterpreted function *M'*.

fun *M* :: *VC* ⇒ *bool* **where**
  M (*VcIf x y*) = ((*M x*) ⟶ (*M y*))
| *M* (*VcAnd x y*) = ((*M x*) ∧ (*M y*))
| *M* (*VcAndAnd x y*) = ((*M x*) ∧ (*M y*))
| *M* (*VcOther x*) = *M' x*

The definition of *M* for *VcAndAnd* could have been written as

$$((\mathrm{M\,x}) \wedge (\mathrm{M\,x} \longrightarrow \mathrm{M\,y})),$$

but in a two-valued logic this is equivalent to the definition given. This is shown by the following lemma.

lemma ((*M x*) ∧ (*M y*)) = ((*M x*) ∧ (*M x* ⟶ *M y*))
  by (*rule iffI*, *simp-all*)

A.4 Auxilliary Lemmas About *foldr* and *map*

To make the proof of the main theorem easier, we first prove some properties about HOL's *foldr* and *map* functions.

It is useful to be able to move one of the conjuncts from the base expression of the *foldr* expression to the outside. This is similar to HOL's *List.foldr_add_assoc*.

lemma *foldr-conj-assoc*:
  shows (*foldr op* ∧ *zs* (*x* ∧ *y*)) = (*x* ∧ (*foldr op* ∧ *zs y*))
    by (*induct zs*) (*simp*, *rule iffI*, *simp-all*)

For the whole *foldr* expression to be *True*, the base expression must be *True*.

lemma *foldr-conj-base*:
shows *foldr* (*op* ∧ ∘ *M*) *xs base* ⟹ *base*
by (*induct xs*) *simp-all*

If the *VcIf*'s antecedant evaluates to *False* then the whole *foldr* expression is *True*.

lemma *foldr-negM*:
shows (¬ *M vc1*) ⟹ *foldr* (*op* ∧ ∘ *M* ∘ *VcIf vc1*) *vc2 True*
by (*induct vc2*) *simp-all*

If the *VcIf*'s antecedant evaluates to *True* then the whole *foldr* expression depends on the value of the consequent.

lemma *M-VcIf-cong*:
shows *M vc1* ⟹ *foldr* (*op* ∧ ∘ *M* ∘ *VcIf vc1*) *vc2 True* = *foldr* (*op* ∧ ∘ *M*) *vc2 True*
by (*induct vc2*) *simp-all*

The simplification procedure can introduce λ expressions, but it is sometimes easier to remove them before proceeding.

lemma *foldr-lambda*:
shows *foldr* (λ*vc*. *op* ∧ (*M vc*)) *vcs base* = *foldr* (*op* ∧ ∘ *M*) *vcs base*
by (*induct vcs*) *simp-all*

foldr-append is defined in HOL's List.thy, but sometimes it is easier to work with an appended list than a nested *foldr*. Care must be taken that the result of applying this lemma not be undone, since *foldr-append* is defined as a *simp* rule.

lemma *foldr-unappend*:
shows *foldr f xs* (*foldr f ys base*) = *foldr f* (*xs* @ *ys*) *base*
by (*induct xs*) *simp-all*

If a *foldr* expresion with the given second parameter evaluates to *True* for a list, it must also evaluate to *True* for sublists. Two particular sublists are of interest.

lemma *foldr-append-left*:
shows *foldr* (*op* ∧ ∘ *M*) (*xs* @ *ys*) *True* ⟹ *foldr* (*op* ∧ ∘ *M*) (*xs*) *True*
by (*induct xs*) *simp-all*

lemma *foldr-append-right*:
shows *foldr* (*op* ∧ ∘ *M*) (*xs* @ *ys*) *True* ⟹ *foldr* (*op* ∧ ∘ *M*) (*ys*) *True*
by (*induct ys*) (*simp*, *rule foldr-conj-base*, *simp*)

A.5 Auxilliary Lemmas for Induction Steps

A few final lemmas are useful to simplify the induction steps in the proofs of the soundness and completeness lemmas.

lemma *mVcIf*:
  assumes *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) *True* ⟹ *M vc1*
      foldr (*op* ∧ ∘ *M*) (*distribImp vc2*) *True* ⟹ *M vc2*
      foldr (*op* ∧ ∘ *M*) (*distribImp* (*VcIf vc1 vc2*)) *True*
  shows *M* (*VcIf vc1 vc2*)
proof (*cases M vc1*)
  case *True*
    thus *M* (*VcIf vc1 vc2*)
     using *assms* **by** (*simp add*: *foldr-map M-VcIf-cong*)
  next case *False*
    thus *M* (*VcIf vc1 vc2*)
     using *assms* **by** (*simp add*: *foldr-map*)
qed

lemma *mVcAnd*:
  assumes *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) *True* ⟹ *M vc1*
      foldr (*op* ∧ ∘ *M*) (*distribImp vc2*) *True* ⟹ *M vc2*
      foldr (*op* ∧ ∘ *M*) (*distribImp* (*VcAnd vc1 vc2*)) *True*
  shows *M* (*VcAnd vc1 vc2*)
proof –
  have *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) (*foldr* (*op* ∧ ∘ *M*) (*distribImp vc2*) *True*)
    using *assms* **by** *simp*
  hence *append*: *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1* @ *distribImp vc2*) *True*
    by (*simp only*: *foldr-unappend*)
  hence *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) *True*
    by (*rule foldr-append-left*)
  hence *M vc1*
    using *assms* **by** *simp*
  moreover have *foldr* (*op* ∧ ∘ *M*) (*distribImp vc2*) *True*
    using *append* **by** (*rule foldr-append-right*)
  hence *M vc2*
    using *assms* **by** *simp*
  ultimately show *M* (*VcAnd vc1 vc2*)
    by *simp*
qed

lemma *mVcAndAnd*:
assumes *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) *True* ⟹ *M vc1*
foldr (*op* ∧ ∘ *M*) (*distribImp vc2*) *True* ⟹ *M vc2*
foldr (*op* ∧ ∘ *M*) (*distribImp* (*VcAndAnd vc1 vc2*)) *True*
shows *M* (*VcAndAnd vc1 vc2*)
proof (*cases M vc1*)
case *True*
hence *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) (*foldr* (*op* ∧ ∘ *M*) (*distribImp vc2*) *True*)
using *assms* **by** (*simp add*: *foldr-map M-VcIf-cong*)
hence *foldr* (*op* ∧ ∘ *M*) (*distribImp vc2*) *True*
by (*simp only*: *foldr-unappend foldr-append-right*)
hence *M vc2*
using *assms* **by** *simp*
thus *M* (*VcAndAnd vc1 vc2*)
using *prems* **by** *simp*
next case *False*
thus *M* (*VcAndAnd vc1 vc2*)
using *assms* **by** (*simp add*: *foldr-map foldr-negM*)
qed

lemma *distribVcIf*:
assumes *M vc1* ⟹ *foldr* (*op* ∧ ∘ *M*) (*distribImp vc1*) *True*
M vc2 ⟹ *foldr* (*op* ∧ ∘ *M*) (*distribImp vc2*) *True*
M (*VcIf vc1 vc2*)
shows *foldr* (*op* ∧ ∘ *M*) (*distribImp* (*VcIf vc1 vc2*)) *True*
proof (*simp add*: *foldr-map*, *cases M vc1*)
case *True*
thus *foldr* (*op* ∧ ∘ *M* ∘ *VcIf vc1*) (*distribImp vc2*) *True*
using *assms* **by** (*simp add*: *foldr-map M-VcIf-cong*)
next case *False*
thus *foldr* (*op* ∧ ∘ *M* ∘ *VcIf vc1*) (*distribImp vc2*) *True*
using *assms* **by** (*simp add*: *foldr-map foldr-negM*)
qed

A.6 Soundness and Completeness

Showing the soundness of our splitting algorithm amounts to showing that if the conjunction of the evaluations of the sub-VCs is true then the original VC evaluates to true. Showing the algorithm's completeness is showing the converse. Thus, showing that the decomposition is both sound and complete is simply showing the implication in both directions.

lemma *soundness*:
  shows *foldr op* $\wedge$ (*map M* (*distribImp vc*)) *True* $\Longrightarrow$ *M vc*
proof (*simp only*: *foldr-map*, *induct vc*)
  case (*VcIf vc1 vc2*)
    thus *M* (*VcIf vc1 vc2*)
     by (*rule mVcIf*)
  next case (*VcAnd vc1 vc2*)
    thus *M* (*VcAnd vc1 vc2*)
     by (*rule mVcAnd*)
  next case (*VcAndAnd vc1 vc2*)
    thus *M* (*VcAndAnd vc1 vc2*)
     by (*rule mVcAndAnd*)
  next case (*VcOther b*)
    thus *M* (*VcOther b*)
     by *simp*
qed

lemma *completeness*:
  shows *M vc* $\Longrightarrow$ *foldr op* $\wedge$ (*map M* (*distribImp vc*)) *True*
proof (*simp only*: *foldr-map*, *induct vc*)
  case (*VcIf vc1 vc2*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcIf vc1 vc2*)) *True*
     by (*rule distribVcIf*)
  next case (*VcAnd vc1 vc2*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcAnd vc1 vc2*)) *True*
     by (*simp add*: *foldr-lambda*)
  next case (*VcAndAnd vc1 vc2*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcAndAnd vc1 vc2*)) *True*
     by (*simp add*: *foldr-lambda foldr-map M-VcIf-cong*)
  next case (*VcOther b*)
    thus *foldr* (*op* $\wedge$ $\circ$ *M*) (*distribImp* (*VcOther b*)) *True*
     by *simp*
qed

theorem *sound-and-complete*:
  shows *foldr op* $\wedge$ (*map M* (*distribImp vc*)) *True* = *M vc*
  using *soundness completeness* ..
end

Appendix B

BISLs and BISL Tools for Java

Several existing projects have added support for DbC to Java, and we provide an overview in this Appendix. To prevent excessive repetition, we note beforehand points of commonality:

- All of the approaches support pre- and postconditions as well as class invariants and a mechanism for accessing the old and return values in postconditions.
- Only RAC is supported, and there is usually little distinction made in the available documentation between the tool and the notation.

Exceptions to these two points will be noted. Summary tables follow the descriptions.

B.1 Jass

Jass was developed as part of Detlef Bartetzko's master's work. It is the most complete of the approaches discussed in this section. In addition to the basic DbC constructs, Jass provides support for quantifiers, simple assertions (checks), loop variants and invariants, rescue blocks with retry for dealing with exceptions, and trace assertions. Behavioral subtyping is optional in Jass, and to enable it, a class must implement a certain interface. Jass is implemented as a preprocessor, and there are 3 levels of severity for a contract violation, including ignoring it, logging it, and throwing an exception [jas, 2007].

B.2 Jcontract and Jtest from Parasoft

Very little detailed information is publicly available from Parasoft. Jcontract is a system for providing DbC for Java. It comes with a replacement for `javac`, called `dbc_javac`, that generates instrumented bytecode. In addition to the standard

DbC annotations, support is provided for concurrency checks, simple assertions, and outputting debug or trace information. The behavior on contract violation can be configured to either log it to a file or to throw an exception. A related product, Jtest provides static enforcement of style rules. It can also generate test cases and test data, and can use Jcontract specifications as an oracle [par, 2009]. A five-seat Server-Edition license costs US$ 50,000 [Coffee, 2006].

B.3 iContract and iContract2

iContract2 is the open-source continuation of iContract, which was developed by the now-defunct Reliable-Systems.com [ico, 2007]. It is used for the Java examples in *Design by Contract, by Example* [Mitchell *et al.*, 2002]. The original source code for iContract no longer exists, and the distributed bytecode was decompiled to restart the project. DbC annotations are included in Javadoc comments and may contain quantifiers and implications. Invariants are not checked for private methods or for `finalize` methods. Contracts from supertypes are checked, but behavioral subtyping is not correctly implemented as preconditions can be strengthened (i.e., an overriding method's precondition is formed from the conjunction of the overridden method's precondition and the one explicitly given). RAC is implemented as a preprocessor that converts annotated Java to instrumented Java before using a standard Java compiler to produce bytecode. iContract2 only supports up to Java 1.4 [iCo, 2001, ico, 2007].

B.4 OVal

OVal is an open-source project that uses Java 5 annotations and AspectJ to implement an unusual variant of DbC. Instead of having explicit pre- and postconditions, member fields and method parameters and non-void return types are annotated with constraints. When methods that are marked with either Pre- or PostValidateThis are called, the constraints are checked at the appropriate time on all fields as well as actual parameters or return value, respectively [ova, 2007].

B.5 Contract4J

Contract4J is an open-source project supported by Aspect Research Associates. Contracts are specified using Java 5 annotations, and AspectJ is used to weave these into the bytecode. Invariants are inherited from supertypes, but not method contracts, which the documentation suggests must be manually copied to subclasses. An empty precondition is shorthand for all of the method's parameters to be non-null [Con, 2008].

B.6 jContractor

jContractor is a recently revived open-source project. Contracts are specified in methods that follow a given naming convention and visibility scheme. These methods return a Boolean value to indicate whether or not their property holds. The framework throws an exception when any of these methods returns false. As an alternative to including contract code inside the class being specified, an additional class containing only contracts can be given. The tool uses reflection to instrument bytecode. Behavioral subtyping seems to be correctly implemented [jCo, 2008].

B.7 C4J

C4J is an open-source project developed by Jonas Bergström that also uses bytecode instrumentation of contracts given in specially named methods and classes. Ensuring behavioral subtyping was one of his motivating goals, but this system seems to be one of the most difficult to use. In addition to the extra classes that must be written, old values must be saved manually in the precondition's method if they are to be accessed in the corresponding postcondition. Simple asserts are used to do the actual checking in the examples given. Invariants are not checked for methods that are marked as pure, but their purity is not verified [c4j, 2007].

B.8 Self-Testable Classes for Java

STclass is a collaborative project of three French universities that combines DbC and unit testing. Originally iContract was used to provide DbC, and while the current version uses the same DbC annotations, it includes an independently developed tool for instrumentation. Also, the testing framework used is not JUnit but their own. Additional annotations are provided for the description of unit tests and suites. During instrumentation, each class is augmented with a `main` method that executes its unit tests. Tests and contracts are inherited from supertypes. The version of the system described in [Jézéquel *et al.*, 2001] makes use of mutation testing techniques to evaluate the quality of a class's test suite, but there is no mention of this in the current documentation [stc, 2007]. There was also earlier mention of automated generation of test data from the specification and using the specifications to generate an oracle [Jézéquel *et al.*, 2001], but again, neither of these is reported in the current documentation [stc, 2007].

Table 5: Status of Java DbC Projects

	Inception	Last Release	Main-tained	Form of Annotations	How Processed
Jass	1999	2005	Yes	in comments	Preprocessor
Jcontract & Jtest			Yes		
iContract	≈2000		No	in Javadoc	Preprocessor
iContract2		2006	Yes	in Javadoc	Preprocessor
OVal	2005	2007	Yes	metadata	
Contract4J	2005	2007	Yes	metadata	
jContractor	1999	2003	No	methods	AspectJ
C4J		2006	Yes	methods	
STclass		2006	Yes		

B.9 Summary

Several existing projects have added support for DbC to Java, but none has yet reached the sophistication of lightweight JML (see Chapter 2.2). Of these, only Jass and Jcontract/Jtest appear to be useful beyond initial dabbling with DbC. Jcontract/Jtest is a proprietary system, so we are unable to consider it for collaboration. There has been talk in recent months about a future version of Jass uniting with JML, but this has mainly been in the context of moving JML specifications from comments to Java metadata annotations, as introduced in Java 5 [mod, 2007]. Java's current annotation facility does not allow for annotations to be located at all syntactic positions where JML annotations can be placed. JSR-308 (Annotations on Java Types) is addressing this problem as a consequence of its mandate, but any changes proposed would only be present in Java 7 [jsr, 2008].

Table 6: Comparison of Features of Java DbC Projects

	Impli-cation	Quati-fiers	Purity	Pre/ Post	Class Inv.	Obj. Inv.	Behavioral Subtyping
Jass	✓	✓	×	✓	×	✓	✓
Jcontract & Jtest	✓	✓		✓	?	✓	✓
iContract	✓	✓	×	✓	?	✓	PreConds anded
iContract2	✓	✓	×	✓	?	✓	PreConds anded
OVal	×	×	×	✓	×	✓	×
Contract4J	×	×	×[a]	✓	?	✓	×
jContractor	×	×	×	✓	×	✓	×
C4J	×	×	×	✓	×	✓	✓
STclass	✓	✓	×	✓	?	✓	✓

[a] A method marked as pure in Contract4J indicates that the class invariant is not checked on entry and exit. This can be automatically detected in some cases.

Table 7: Examples of Annotations from Java DbC Projects' Documentation

	Sample Annotation
Jass	`@ensure !isEmpty(); Old.count = count-1;`
Jcontract & Jtest	`@ensure return != null`
iContract	`@post b@pre implies return > 0`
iContract2	`@post b@pre implies return > 0`

OVal	`@post(expr="_this.amount > _old",`
	`old="_this.amount",`
	`lang="groovy")`

Contract4J	`@Post("$return != null")`

jContractor	`protected boolean size_`
	`Postcondition(int Result) {`
	`return Result >= 0;`
	`}`

C4J	see Figure 44.
STclass	`@post return implies hasExits()`

```
@ContractReference(contractClassName = "DummyContract")
public class Dummy
{
  protected List m_stuff = new LinkedList();

  public int addItem(Object item)
  {
    m_stuff.add(item);
    return m_stuff.size();
  }
}

public class DummyContract extends ContractBase<Dummy>
{
  public DummyContract(Dummy target)
  {
    super(target);
  }

  public void classInvariant()
  {
    assert m_target.m_stuff != null;
  }

  public void pre_addItem(Object item)
  {
    super.setPreconditionValue("list-size", m_target.m_stuff.size());
  }

  public void post_addItem(Object item)
  {
    assert m_target.m_stuff.contains(item);
    int preSize = super.getPreconditionValue("list-size");
    assert preSize == m_target.m_stuff.size() - 1;
    assert m_target.m_stuff.size() == super.getReturnValue();
  }
}
```

Figure 44: Example of C4J Annotation

Appendix C

SPARK

SPARK [Barnes, 2006] was developed by Program Validation Limited (later acquired by Praxis Critical Systems Limited) for the implementation of safety-critical avionics and rail-control systems. It has now been used in other domains where high integrity is required. These include "finance, communications, medicine", and automotive systems. SPARK is a subset of Ada extended with annotations in comments that increase the expressiveness of interfaces and provide support for DbC. The subset was chosen to be amenable to ESC and FSPV yet useful for writing industrial applications. A methodology is provided that helps ensure the correctness of systems developed with SPARK.

Many Ada constructs are not supported in SPARK, such as implementation dependent constructs. SPARK systems, like those in Ada, are made from packages and subprograms. Two kinds of subprograms are distinguished based on whether they return a value: functions do, and procedures do not. SPARK functions are not allowed to have side effects. Abstract Data Types (ADTs) can be defined and extended, but polymorphism and dynamic dispatch are not supported because of the difficulty of statically reasoning about them. Generics, pointers, unchecked casting, exceptions, overloading, and aliasing are similarly banned. Recursion and dynamic storage allocation from the heap are forbidden so that an upper bound can be statically determined for the time and memory requirements of a system.

Despite these limitations, SPARK has many useful features including allowing compound types (i.e., records or structures) as function return types and unconstrained (i.e., unbounded) array types as parameters. SPARK also has some features not found in all mainstream languages, such as enumerated types and subranges of numeric and enumerated types.

There are two kinds of annotations in SPARK:

- **Core** annotations for flow analysis and visibility control and
- **Proof** annotations for expressing contracts and to guide proof tools.

```
Procedure Add(X: in Integer);
--# global  in out Total, Grand_Total;
--# derives Total from Total, X &
--#          Grand_Total from Grand_Total, X;
```

Figure 45: SPARK example showing core annotations

Quantifiers over numerics and enumerations are provided with the keywords `for all` and `for some`. Proof functions, similar to JML's model methods, can be defined and used in annotations.

In Ada, parameters are marked with one of three *modes*:

- `in` — the parameter may be read, but should not be modified
- `out` — the parameter may be modified, but should not be read
- `in out` — the parameter may be both read and modified.

SPARK changes the meaning of these by replacing *may* with *must* and *should not* with *must not*. Similarly, global (i.e., package-level) variables that are accessed in a subprogram must be included in its header and given a mode. Further, a `derives` clause can be given that shows which variables are used in the computation of `out` and `in out` parameters and globals. An example from [Barnes, 2006] showing the use of these constructs is shown below.

Since SPARK is a proper subset of Ada, any standard compiler can be used. Unlike the other languages discussed in this in Section 2.3.1, there is no support for RAC, since static verification is used to show that errors, including contract violations, cannot happen at runtime. Tools provided by Praxis include the

- Examiner
- Simplifier
- Proof Checker
- Proof Obligation Summarizer (POGS).

The Examiner and POGS are written in SPARK itself, and the Simplifier and the Proof Checker are written in Prolog.

In addition to syntax and type checking and ensuring that only constructs in the SPARK subset are present, the **Examiner** is used to provide three levels of verification rigor. The first is data-flow analysis, in which it checks that parameters and globals conform to their declared modes, that variables are not read before being initialized, and that all assigned variables are later used. Information-flow analysis checks if `derives` clauses are correct. Finally, the Examiner can produce a `.vcg` file that contains VCs to check the remaining annotations, and these can

be discharged either formally by the Simplifier and Proof Checker or manually by human reviewers.

As with ESC for JML and Spec#, subprogram calls are replaced with specifications and not implementations. The Examiner handles loops in a manner similar to that of Spec# (see above), but it does not do any checking for termination. The Examiner is able to detect and warn if a loop is stable (i.e., if none of the variables in the condition are modified in the body).

The **Simplifier** incorporates an automated theorem prover that usually discharges most of the VCs generated by the Examiner. The VCs related to run-time errors, such as array-bounds errors, are always simple enough for it to verify. Many of the VCs related to contracts are also trivial, but this group may contain some that are beyond its reasoning ability. The Simplifier's output is a log file and an `.siv` file, which contains unproved VCs.

The **Proof Checker** is an interactive theorem prover that can be used to discharge the VCs left by the Simplifier. As with other FSPV tools, the user must guide the proof, while the tool keeps track of subgoals that remain to be proved. Users can extend the set of strategies and rules used by the Proof Checker.

The **POGS** tool is used to reduce the various outputs of the other three tools to a single report that gives the status of verification process, including a list of any VCs that remain unproved.

Appendix D

RAC-ing ESC/Java2[1]

This Appendix presents the details of a case study in which we compiled and ran a RAC-enabled version of ESC/Java2. Section D.1 describes the compilation of the ESC/Java2 source with the JML RAC. Section D.2 covers the main design issues in ESC/Java2 that have been uncovered by using the JML RAC.

D.1 Origin of the Case Study

The case-study was initiated in the summer of 2005 by Dr. Patrice Chalin and Dr. Joseph Kiniry. Prior to that time, the ESC/Java2 source had not been compiled with the JML RAC. Because of the historical differences in input languages between ESC/Java2 and the JML Checker and RAC, it took approximately four developer-weeks for them to make the necessary updates to source code of ESC/Java2 for it be acceptable to the Checker. This makes the ESC/Java2 source able to be compiled with the RAC. Most of these changes were due to slight incompatibilities between the syntax accepted by the JML compiler and that used in the source files. A few bugs were removed from (or, more accurately, enhancements were made to) the repository of API specifications (e.g., `java.lang`, `java.lang.util`, etc.) that are distributed with ESC/Java2.

The exercise also allowed them to uncover and file reports on 8 bugs in the JML RAC. A major problem, which was not resolved until later by Frédéric Rioux as part of his Master's research, prevented the JML RAC from creating instrumented `.class` files for three classes because the checking code had a `try/catch` block that is larger than the limits allowed by the JVM [Rioux, 2006]. With this problem overcome, we were able to compile the 550 classes of the ESC/Java2 application with `jmlc` in a little over 7 minutes (on a 2 GHz P4). These classes are distributed over 3 main packages:

- `javafe`, a common front end used by ESC/Java2 and other tools, such as Houdini [Flanagan and Leino, 2001].

[1] This appendix is based on [Chalin and James, 2006].

- `escjava`, a package that builds on the services provided by the Javafe to implement the extended static checking functionality.
- `junitutils`, various support utilities, particularly with automated testing.

D.2 Compiling ESC/Java2 Source with the JML RAC

In the subsections that follow, we detail some of the major problems reported by ESC/Java2 when running the RAC compiled version. Note that all of these errors were identified during static initialization. That is, these errors report inconsistencies between the static initialization code and the JML specifications of the ESC/Java2 classes. The errors are presented essentially in the order in which they were discovered. As was mentioned above, we will see that the errors identified have fairly deep design implications.

D.2.1 AST Node Invariants Not Established by Constructors

The first error to be reported by the RAC-instrumented ESC/Java2 is shown in Figure 46. We see that the class invariant (`JMLInvariantError`) of the `PrimitiveType` class was violated on exit (i.e., during the verification of the postcondition) of the `PrimitiveType` constructor (represented by `¡init¿`). We will soon see why ESC/Java2 did not report this error.

The violation occurred at line 128 of the file `Primitivetype.java`. This file is part of the collection of `javafe` Abstract Syntax Tree (AST) node classes. An excerpt of the file is given in Figure 47. The figure shows only one sample invariant clause (constraining the value of the `tag` field) at the start of the file. A static `make()` method and the problematic constructor are also shown. At line 128, we see that the body of `PrimitiveType()` is empty. Its associated Javadoc comment explains why. Apparently, a fundamental design decision for the AST node class hierarchy had been to have all AST nodes created via maker methods (generally named `make()`). The maker methods first invoke a default constructor having an empty body and then proceed to initialize the object fields. The AST node instance returned by this maker method is meant to satisfy its class invariant.

Of course, class invariants are meant to hold for *all* instances of the class including those created by default constructors with empty bodies. Since this is clearly not the case for the AST node constructors, use is made of the ESC/Java2 `nowarn` pragma. This pragma allows developers to instruct ESC/Java2 to ignore certain kinds of errors—e.g., invariant errors in the case of `PrimitiveType`. As a reminder to developers of the obligation to establish the class invariant after calling the default constructor, a specification-only (ghost) variable named "`I_will_establish_invariants_afterwards`" was created. We see in Figure 47 how the `make()` method uses the default constructor, sets the instance fields, and sets the special-purpose ghost variable to `true`. While there are a number of solutions

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInvariantError:
by method PrimitiveType.¡init¿@post
File "Javafe/java/javafe/ast/PrimitiveType.java", line 128, character 15
regarding specifications at
File "Javafe/java/javafe/ast/PrimitiveType.java", line 35, character 17 when
'tag' is 0
'this' is [PrimitiveType tmodifiers = null tag = 0 loc = 0]
at javafe.ast.PrimitiveType.checkInv$instance$PrimitiveType (PrimitiveType.java:958)
at javafe.ast.PrimitiveType.¡init¿(PrimitiveType.java:210)
at javafe.ast.PrimitiveType.internal$makeNonSyntax(PrimitiveType.java:97)
at javafe.ast.PrimitiveType.makeNonSyntax(PrimitiveType.java:3029)
at javafe.tc.Types.internal$makePrimitiveType(Types.java:154)
at javafe.tc.Types.makePrimitiveType(Types.java:4016)
at javafe.tc.Types.¡clinit¿(Types.java:19)
at escjava.Main.¡init¿(Main.java:78)
at escjava.Main.compile(Main.java:215)
at escjava.Main.main(Main.java:177)
```

Figure 46: Run-time assertion violation reported by ESC/Java2 compiled with the JML RAC

to this problem, the simplest was to eliminate the default constructor in favor of constructors that establish invariants right from the start. In doing so, we simplified the design by consolidating the instance creation process, eliminating the `I_will_establish_invariants_afterwards` variable and the `nowarn` pragma. In this way, both ESC/Java2 and the JML RAC can process the resulting specifications. While our new design impacted almost two hundred classes, most of the changes were confined to AST node generation routines and templates.

We note that there are generally two main reasons for using `nowarn` pragma:

1. When a specifier believes something to be true but the verifier is unable to confirm its truth. In such a case, the RAC facility can confirm that the specification does indeed hold at runtime for the exercised test cases.

2. When a specifier knows something to be false, but wants to ignore it for the moment and continue making progress (in verifying other parts of the program). The RAC will catch these violations and prevent the system from being usable.

It would be helpful to developers if all `nowarn`s were commented with the reason for their presence. Instances of (2) should be resolved as quickly as possible so that all of our tools can be used in support of our development efforts. It would appear that the case treated in this subsection is an instance of (2)—maybe there was a belief that the use of non-default constructors was not feasible, when in fact it turns out to be straightforward.

```
public class PrimitiveType extends Type
{
  /*@ invariant (tag == TagConstants.BOOLEANTYPE || ...); */
  public int tag;

  //@ requires (tag == TagConstants.BOOLEANTYPE || ...);
  //@ ensures ...
  public static /*@ non_null */ PrimitiveType
      make(TypeModifierPragmaVec tmodifiers, int tag, int loc)
  {
      //@ set I_will_establish_invariants_afterwards = true;
      PrimitiveType result = new PrimitiveType();
      result.tag = tag;
      result.loc = loc;
      result.tmodifiers = tmodifiers;
      //...
      return result;
  }

  /**
   * Construct a raw PrimitiveType whose class invariant(s) have not
   * yet been established.  It is the caller's job to initialize the
   * returned node's fields so that any class invariants hold.
   */
  //@ requires I_will_establish_invariants_afterwards;
  protected PrimitiveType() {}    //@ nowarn Invariant,NonNullInit; // ** LINE 128 **
  ...
}
```

Figure 47: Excerpt from `javafe/ast/PrimitiveType.java`

D.2.2 Internal AST Node Instances vs. AST Node Class Invariants

The next two problems reported by the RAC are related to the creation of an internal field for the length of arrays (namely, `lengthFieldDecl`), itself of type `int` (Figure 48). The violations were, firstly, of the invariant of `GenericVarDecl` that `type.syntax` be `true` (i.e., that the type be an AST node read from a file, not an internally created type like `Types.intType`)—see Figure 49. The second violation had to do with the `locId` of the `FiedlDecl` `maker` method: it was required to be different from `Location.NULL`. Of course, neither of these conditions is satisfied by the call to `maker()` in Figure 48.

After some analysis, and two unsatisfactory attempts, the solutions we implemented was to create a new `maker` method and constructors for `FieldDecl` and

```
public static /*@ non_null */ FieldDecl lengthFieldDecl
   = FieldDecl.make(..., lenId, Types.intType, Location.NULL,...);
```

Figure 48: Declaration of `length` field for arrays in `javafe.tc.Types`

```
package javafe.ast;

public abstract class GenericVarDecl extends ASTNode
{
  ...
  public /*@ non_null @*/ Type type;
  //@ invariant type.syntax;
  ...
}

public class FieldDecl extends GenericVarDecl implements ...
{
  ...
  //@ requires locId != javafe.util.Location.NULL;
  //@ ensures \result != null;
  public static FieldDecl make(...,
                               /*@ non_null @*/ Type type,
                               int locId, ...)
  {
     //...
  }
}
```

Figure 49: Excerpts from `GenericVarDecl` and `FieldDecl` of `javafe.ast`

`GenericVarDecl` that do not take a location. These would set the `GenericVarDecl`'s `locId` to `Location.NULL`. To capture the idea of an internal field, an `isInternal()` method was added to `GenericVarDecl`. This method returns `true` exactly when the location is not equal to `Location.NULL`. Because of these changes, `FieldDecl`'s new maker method no longer mentions location, and the old one remains unchanged. The invariant of `GenericVarDecl`, `FieldDecl`'s super class, was changed to reflect that `syntax` is `true` exactly when `isInternal()` is `false`. To reflect that there are no internal AST nodes of subclasses other than `GenericVarDecl` (namely,

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLInternalPreconditionError:
  by method PrimitiveType.makeNonSyntax regarding specifications at
File "Javafe/java/javafe/ast/PrimitiveType.java", line 78, character 16 when
        'tag' is 247
        at escjava.Main.compile(Main.java:4138)
        at escjava.Main.internal$main(Main.java:118)
        at escjava.Main.main(Main.java:3479)
```

Figure 50: RAC error: violation of `PrimitiveType maker` method precondition

`FormalParaDecl` and `LocalVarDecl`), all other AST node classes have `!isInternal()` as an invariant.

D.2.3 Specification and Polymorphic Structures

A very interesting design problem that runtime assertion checking highlighted involved (a violation of) behavioral subtyping [Liskov and Wing, 1994]. As mentioned above, Java's primitive types are represented using instances of PrimitiveType. This class belongs to the javafe package, which is common to tools that need a Java front end. Primitive types are distinguished by a tag attribute. The maker methods require that a valid tag be used when creating a new instance of PrimitiveType, and an invariant ensures that the tag remains valid. A valid tag is defined in PrimitiveType to be one of ten given tag values. The tags themselves are defined in the class `javafe.ast.TagConstants`.

As was mentioned earlier, the `escjava` package makes use of services of the Java front-end package. In particular, it makes direct use of the `PrimitiveType` class to define ESC/Java2- and JML-specific primitive types such as `lockset` and `\bigint`. To do so, new tags are defined in `escjava.ast.TagConstants`. Unfortunately, the static creation of, e.g., the `escjava lockset` primitive type results in a violation of the `PrimitiveType maker` method's precondition (see Figure 50) since the `maker` is given a `tag` value that is not one of the expected ten "valid" values.

One approach considered was to define a subtype of `javafe.ast.PrimitiveType` named `escjava.ast.EscPrimitiveType` and represent the ESC/Java2 and JML primitive types with instances of this new class. Unfortunately, the semantics of class invariants and the enforcing of behavioral subtyping in JML make it impossible to write any useful class contracts for `PrimtiveType` and `EscPrimitiveType` in such a case (even if, for example, we use an auxiliary boolean method `isValidTag`). The problem is illustrated in Figure 51. The first problem to be noticed is that it is difficult to choose an appropriate class invariant restricting the value of `tag`. E.g., it cannot be limited to only `javafe` tags, otherwise `EscPrimitiveTypes` could not be created. We cannot say in `javafe.ast.PrimitiveType` that the legal tags also include those of the `escjava.ast` package since this would create circular dependencies between `javafe` and `escjava`. Similarly, notice how the

```
package javafe.ast;
public class PrimitiveType extends Type {
  //@ invariant (tag == TagConstants.BOOLEANTYPE || ...); // ???
  public int tag;

  //@ requies ???
  protected PrimitiveType(\ldots, int tag, int loc) {
     this.tag = tag;
     ...
  }
}

package escjava.ast;
public class EscPrimitiveType extends PrimitiveType
{
  //@ requires (* tag is a valid javafe tag or an esc tag *);
  protected EscPrimitiveType(\ldots, int tag, int loc)  {
       // tag might not be a valid PrimitiveType tag!
       super(tmodifiers, tag, loc);
  }
  // ...
}
```

Figure 51: Sample (invalid) solution: excerpts from `PrimitiveType` and `EscPrimitiveType`

`EscPrimitiveType` constructor invokes the `PrimitiveType` constructor (via `super`). For this call to be permitted, what precondition must the `PrimitiveType` constructor have with respect to its `tag` parameter?

Instead of trying to work with the untenable solution consisting of two classes, we decided to extract an abstract superclass and allow both the Java front end and ESC tools to implement the abstract method defined in this superclass while inheriting the rest of its functionality. The superclass has both a code and model version of an `isValidTag` method. By specifying that the code version's result is the same as the model version's, we are able to statically verify the invariant that `isValidTag` always returns `true`.

The two concrete subclasses (namely, `JavaFePrimitiveType` and `EscPrimitiveType`) have implementations of isValidTag that compare against the appropriate values in each case. Their makers and constructors require that the tag value passed to them be valid, as determined by their local versions of isValidTag. Since the value passed to the makers and constructor is valid, and since this value is

```
package javafe.ast;
public abstract class PrimitiveType
{
  //@ public model instance int _tag;
  //@ pure model boolean specIsValidTag(int tag);

  //@ ensures \result == specIsValidTag(_tag);
  /*@ pure */ public abstract boolean isValidTag();

  //@ public invariant isValidTag();

  //@ ensures \result == _tag;
  /*@ pure */ public abstract int getTag();
}
```

Figure 52: Excerpt of correct redesign of `PrimitiveType`, part 1

stored as the type's tag, the invariant can be statically shown to hold. This solution is illustrated in Figures 52 and 53.

D.2.4 Internal `Literal` Instances vs. `Literal` Class Invariants

The next reported by the RAC was similar to previous errors in which the class had an invariant that was too strict. In this case, the `LiteralExpr` used to define the internal constants for `true` and `false` violated the invariant that their `location` be valid. Since these values are not defined in code, they do not have a valid `location`. This appeared to be a valid use of `LiteralExpr`, so the specification was taken to be incorrect.

Instead of removing altogether the offending requirement that the `location` never equal the constant `Location.NULL`, a second `maker` (namely, `makeNonSyntax`, which follows the naming used in `PrimitiveType`) was added that does not take a `location`. This is compatible with common practice of introducing factory methods for the same type but building objects for different purposes [Bloch, 2001, item 1].

```
package escjava.ast;
public class EscPrimitiveType extends PrimitiveType
{
  /*@ spec_public */ private int tag;
  //@ public represents _tag <- this.tag;

  /*@ public normal_behavior
    @    ensures \result == (JfePrimitiveType.isValidTag(tag) ||
    @            tag == TagConstants.LOCKSET || ...);
    @*/
  public static /*@pure*/ boolean isValidTag(int tag) {
     return (JfePrimitiveType.isValidTag(tag) ||
             tag == TagConstants2.LOCKSET || ...);
  }

  /*@ also
    @ public normal_behavior
    @    ensures \result == EscPrimitiveType.isValidTag(tag);
    @*/
  public /*@pure*/ boolean isValidTag() {
     return isValidTag(tag);
  }

  /*@ public normal_behavior
    @    ensures \result == EscPrimitiveType.isValidTag(tag);
    @ public model pure boolean specIsValidTag (int tag) {
    @     return EscPrimitiveType.isValidTag(tag);
    @ }
    @*/

  /*@ protected normal_behavior
    @    requires EscPrimitiveType.isValidTag(tag);
    @    ensures  this.tag == tag && ...;
    @*/
  protected /*@pure*/ EscPrimitiveType(..., int tag, int loc)  {
     this.tag = tag; ...
  }
}
```

Figure 53: Excerpt of correct redesign of `PrimitiveType`, part 2

from `LiteralExpr.java`

```
//@ requires loc != javafe.util.Location.NULL;
public static /*@non_null*/ LiteralExpr make(int tag, Object
                                                value, int loc) {
```

from `AnnotationHandler.java`

```
public final static LiteralExpr T = (LiteralExpr)
        FlowInsensitiveChecks.setType(LiteralExpr.make(
                                TagConstants.BOOLEANLIT, Boolean.TRUE,
                                Location.NULL),
            Types.booleanType);
```

Figure 54: Definition of `javafe.ast.LiteralExpr`'s `maker` and a call to it

Appendix E

Early Validation: Non-null Type System[1]

The Section 3.2 presented the architecture of JML4. In this Appendix we show how we used its core functionality, including its ability to use JML API library specifications and the Non-Null Type System, to annotate and type check a non-trivial amount of Java source code. This allowed us to gather statistics in support of Dr. Chalin's proposal that reference types be non-null by default.

We conducted an empirical study of 5 open-source projects totaling 700 KLOC that confirmed the hypothesis that on average 75% of reference declarations are meant to be non-null by design. Guided by these results, Dr. Chalin proposed the adoption of a non-null-by-default semantics. This new default has advantages of better matching general practice, lightening developer annotation burden, and being safer. This new default was implemented in JML4, which supports the new semantics and can read the extensive API library specifications written in the Java Modeling Language (JML). In a second phase of the empirical study, we analyzed the uses of null and noted that over half of the nullable field references are only assigned non-null values. Details of this second part are discussed in Section 7.

E.1 Motivation

One of JML4's first and most fully developed features was support for JML's non-null type system [Chalin and James, 2007]. This, coupled with the tool's ability to read the extensive JML API library specifications, renders it quite effective at statically detecting potential null-pointer exceptions (NPEs). Early on, JML4 was enhanced to support Extended Static Checking (ESC) through the integration of ESC/Java2 [Cok and Kiniry, 2005]. While each verification technique has strengths

[1]This appendix is based on [Chalin *et al.*, 2008c].

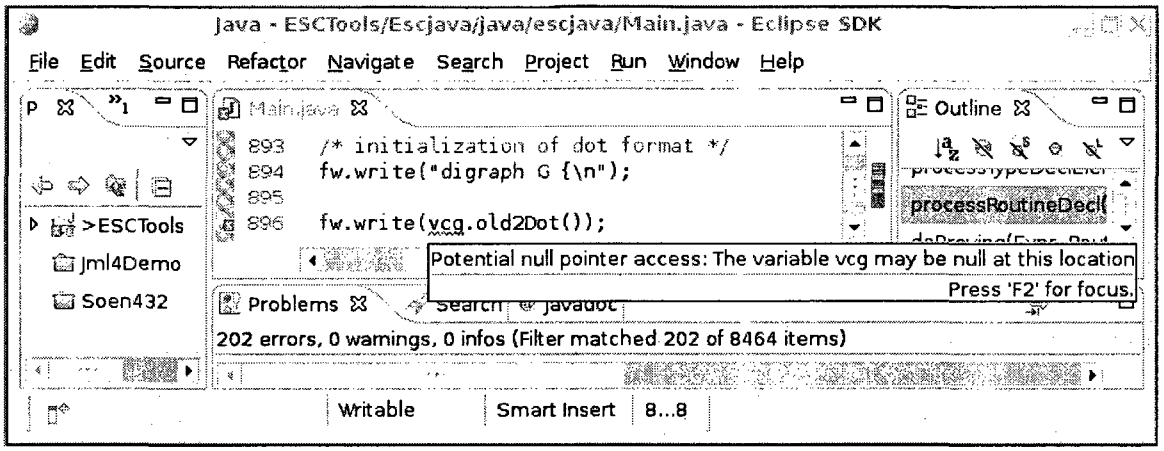

Figure 55: JML4 reporting non-null type system errors in a method too big for ESC/Java2 to verify

and weaknesses, the integration of complementary techniques into a single verification environment brought about a level of synergy that was not otherwise be achievable.

As a concrete example of the kind of verification-technique synergy that JML4 achieves, consider the code fragment given in Figure 55, an excerpt from ESC/Java2's `escjava.Main` class. JML4 correctly reports that a dereference of `vcg` in `processRoutineDecl()` could result in an NPE.

ESC/Java2 is routinely run on itself, but this error was not detected before because analyzing `processRoutineDecl()`, whose body has 386 lines of code, is beyond the capabilities of ESC/Java2. It gives up on attempting to verify the method because the verification condition generated for it is too big. Several errors that arise under similar circumstances were identified in ESC/Java2 source by JML4.

As another example, consider the static `options()` method of `escjava.Main`, which returns a reference to ESC/Java2's command-line options (see Figure 56). This method is used throughout the code (272 occurrences), and its return value is directly dereferenced even though the method can return `null`.

JML4 reports the 250+ NPEs related to the use of this method, but ESC/Java2 does not because another detected error prevents it from determining that the method can return `null`. In this particular case, it is a possible type-cast violation. ESC/Java2 is more susceptible than ordinary compilers to the effects of one error masking others. This makes the more resilient, though less powerful, complementary verification capabilities of other techniques, such as those implemented in JML4, more effective.

Our preliminary use of JML4 demonstrated that fixing some kinds of errors (e.g., nullity type errors) allows ESC/Java2 to push its analysis further, helping expose yet more bugs in code and specifications. This leads to uncovering further nullity type errors, and the process iterates.

```
package escjava;
...
public class Main extends javafe.SrcTool {
    ...
    public static Options options() {
        return (Options)options;
    }
    ...
    public String processRoutineDecl(...) {
        ...
        VcGenerator vcg = null; ...
        try {
            ... // possible assignment to vcg
        } // multiple catch blocks
        catch (Exception e) {
            ...
        }
        ...
        fw.write(vcg.old2Dot()); // <<< possible NPE
        ...
    }
}
```

Figure 56: Code except from the `escjava.Main` class

E.2 The Case Study

Table 8 provides the number of files, lines of code (LOC) and source lines of code (SLOC) [Park, 1992] for our study subjects as well as the projects of which they are subcomponents.

E.2.1 Verification and Validation of Annotations

We used two complementary techniques to ensure the accuracy of the nullity annotations that we added. First, we compiled each of the study subjects using JML4 with RAC enabled and then ran it against each project's standard test suite. Nullity RAC ensures that a `non-null` declaration is never initialized or assigned `null`, be it for a local variable, field, parameter, or method return declaration. In some cases, the test suites are quite large (e.g., on the order of 15,000 tests for the Eclipse JDT, 50,000 for JML, and 600 for ESC/Java2). While the number of tests for ESC/Java2 is lower, some of the individual tests are big (e.g., the type checker

Table 8: General statistics of study subjects and their encompassing projects

Encompassing Project →	Common JML Tools	ESC Tools	SoenEA	Koa	Eclipse JDT	Total
# of files	831	455	52	459	4124	5921
LOC (K)	243	124	3	87	1018	1475
SLOC (K)	140	75	2	62	660	939
Study Subject →	**JML Checker**	**ESC/ Java2**	**SoenEA**	**Koa Tally Subsystem**	**Eclipse JDT Core**	**Total**
# of files	217	216	52	29	1130	1644
LOC (K)	86	63	3	10	560	722
SLOC (K)	58	41	2	4	365	470

is run on itself). In addition, we ran the RAC-enabled version of ESC/Java2 (i.e., a version that performed runtime checks of ESC/Java2's nullity annotations) on all files in the study samples; the increased number of checks of ESC/Java2's nullity annotations increased our confidence in their correctness. Though testing can provide some level of assurance, coverage is inevitably partial and depends highly on the scope of the test suites.

When applying the second technique, we also made use of the ESC/Java2 static analysis tool. In contrast to runtime checking, static analysis tools can verify the correctness of annotations for "all cases" (within the limits of the completeness of the tool), but this greater completeness comes at a price: In many cases, general method specifications (beyond simple nullity annotations) needed to be written to eliminate false warnings.

Using these techniques we were able to identify about two dozen (0.9%) incorrectly annotated declarations—excluding errors we corrected in files outside of the sample set. With these errors fixed, tests passing, and ESC/Java2 not reporting any nullity warnings, we are very confident of the accuracy of the final annotations.

E.2.2 Statistics Tool

To gather statistics concerning non-null declarations, we created a simple Eclipse JDT abstract syntax tree (AST) visitor which walks the Java AST of the study subjects and gathers the required statistics for relevant declarations. A previous attempt at this study made use of an enhanced version of the JML checker which both counted and inferred nullity annotations using static analysis driven by elementary heuristics. For our work, we decided instead to annotate all declarations explicitly and use a simple visitor to gather statistics. This helped us eliminate

one threat to internal validity that arose due to completeness and soundness issues of the enhanced JML-checker-based statistics-gathering feature.

E.3 Study Results

A summary of the statistics of our study samples is given in Table 9. As is customary, the number of files in each sample is denoted by n and the population size by N. Note that for `SoenEA`, 11 of the files did not contain any declarations of reference types, so the population size is $41 = 52 - 11$. We exclude such files from our sample because it is not possible to compute the proportion of non-null references for files without any declarations of reference types. We see that the total number of declarations that are of a reference type (d) across all samples is 2839. The total number of such declarations constrained to be non-null (m) is 2319. The proportion of non-null references across all files is 82%.

We also computed the mean, $\bar{x}$, of the proportion of non-null declarations on a per file basis ($x_i = m_i/d_i$). The mean ranges from 79% for the Eclipse JDT Core, to 89% for the JML checker. Also given are the standard deviation (s) and a measure of the maximum error (E) of our sample mean as an estimate for the population mean with a confidence level of $1 - \alpha = 95\%$. The overall average and weighted average (based on N) for μ_{min} are 80% and 74%, respectively. With this, we can conclude with 95% certainty that the population means are above $\mu_{min} =$ 74% in all cases. As explained earlier, we were conservative in our annotation exercise, hence it is quite possible that the actual overall population mean is greater than this.

We conclude that the study results clearly support the hypothesis that in Java code, over 2/3 of declarations that are of reference types are meant to be non-null. In fact, it is closer to 3/4.

Table 9: Distribution of the number of declarations of reference types

	JML Checker	ESC/ Java2	SoenEA	Koa TS	Eclipse JDT Core	Sum or Average
n	35	35	41	29	35	175
N	217	216	41	29	1130	1633
$\sum d_i$	420	989	231	566	633	2839
$\sum m_i$	362	872	196	424	465	2319
$\sum m_i / \sum d_i$	86%	88%	85%	75%	73%	82%
mean ($\bar{x}$)	89%	85%	84%	80%	79%	83%
std.dev.(s)	0.14	0.22	0.28	0.26	0.24	-
E (α=5%)	4.4%	6.8%	-	-	7.7%	-
$\mu_{min} = \bar{x} - E$	85%	78%	84%	80%	71%	80%

E.4 Summary

In this section, we report on a novel study of five open projects (totaling over 722 KLOC) taken from various application domains. The study results show that on average, one can expect approximately 75% of reference type declarations to be non-null by design in Java. We believe that this report was originally made at a timely point, as we are witnessed the increasing emergence of static analysis (SA) tools using non-null annotations to detect potential null-pointer exceptions. Before too much code is written under the current nullable-by-default semantics, it would be preferable that Java be adapted, or at least a *standard* non-null annotation-based extension be defined, in which declarations are interpreted as non-null by default. This would be the first step in the direction of an apparent trend in the modern design of languages with pointer types, which is to support non-null types and non-null by default [Chalin *et al.*, 2008c].

One might question whether a language as widely deployed as Java can switch nullity defaults. If the successful transition of Eiffel is any indication, it would seem that the switch can be achieved if suitable facilities are provided to ease the transition. We believe that JML4 offers such facilities in the form of support for project-specific as well as fine-grained control over nullity defaults (via type-scoped annotations). Until standard (Java 5) nullity annotations are adopted via JSR 305, we have designed JML4 to recognize JML-style nullity modifiers, thus allowing the tool to reuse the comprehensive set of JML API specifications (among other advantages). Adding nullity annotations is time consuming. By adopting JML-style nullity modifiers we also offer developers potentially increased payback, in that all other JML tools will be able to process the annotations as well—including the SA tool ESC/Java2 and JmlUnit, which generates JUnit test suites using JML specifications and annotations as test oracles.