Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)

SIGSOFT 2008/FSE 16
16th ACM SIGSOFT Symposium on the Foundations of Software Engineering
Atlanta, Georgia, USA
November 9-10, 2008

Technical Report CS-TR-08-07,
School of Electrical Engineering and Computer Science, University of Central Florida,
4000 Central Florida Blvd., Orlando, Florida, 32816-2362 USA

SAVCBS 2008 PROCEEDINGS

Specification and Verification of Component-Based Systems

http://www.eecs.ucf.edu/SAVCBS/

November 9-10, 2008
Atlanta, Georgia, USA

Workshop at SIGSOFT 2008/FSE 16
16th ACM SIGSOFT Symposium on the
Foundations of Software Engineering

SAVCBS 2008
TABLE OF CONTENTS

SAVCBS 2008
ORGANIZING COMMITTEE

Jonathan Aldrich (Carnegie Mellon University, USA)
Jonathan Aldrich is an assistant professor in the School of Computer Science at Carnegie Mellon University. His research contributions include techniques for verifying object interaction protocols and architectures, modular reasoning techniques for aspects and stateful programs, and new object-oriented language models. He received his Ph.D. from the University of Washington in 2003.

Mike Barnett (Microsoft Research, USA)
Mike Barnett is a Research Software Design Engineer in the Foundations of Software Engineering group at Microsoft Research. His research interests include software specification and verification, especially the interplay of static and dynamic verification. He received his Ph.D. in computer science from the University of Texas at Austin in 1992.

Dimitra Giannakopoulou (RIACS/NASA Ames Research Center, USA)
Dimitra Giannakopoulou is a RIACS research scientist at the NASA Ames Research Center. Her research focuses on scalable specification and verification techniques for NASA systems. In particular, she is interested in incremental and compositional model checking based on software components and architectures. She received her Ph.D. in 1999 from the Imperial College, University of London.

Gary T. Leavens (School of EECS, University of Central Florida, USA)
Gary T. Leavens is a professor in the School of Electrical Engineering and Computer Science at the University of Central Florida. He moved to Orlando in Fall 2007. Previously he was a professor of Computer Science at Iowa State University. His research interests include programming and specification language design and semantics, program verification, and formal methods, with an emphasis on the object-oriented and aspect-oriented paradigms. He received his Ph.D. from MIT in 1989.

Natasha Sharygina (CMU and SEI, USA; Lugano, Switzerland)
Natasha Sharygina is a senior researcher at the Carnegie Mellon Software Engineering Institute and an adjunct assistant professor in the School of Computer Science at Carnegie Mellon University, and an assistant professor at the University of Lugano. Her research interests are in program verification, formal methods in system design and analysis, systems engineering, semantics of programming languages and logics, and automated tools for reasoning about computer systems. She received her Ph.D. from The University of Texas at Austin in 2002.

SAVCBS 2008
PROGRAM COMMITTEE

Robby (Department of Computing and Information Sciences, Kansas State University, USA)
Robby chaired the program committee for SAVCBS 2008. He is an assistant professor in the Department of Computing and Information Sciences, Kansas State University. His research interests are in software specification, analysis, transformation, and model-driven software development. He received his Ph.D. in Computer Science from Kansas State University in 2004.

Workshop Program Committee:
Patrice Chalin (Concordia University, Canada)
Ivica Crnkovic (Mälardalen University, Sweden)
Cormac Flanagan (University of California, Santa Cruz, USA)
Alex Groce (Jet Propulsion Laboratory, USA)
Joseph Kiniry (University College Dublin, Ireland)
Eric Madelaine (INRIA, Sophia Antipolis, France)
Rupak Majumdar (UCLA, USA)
Darko Marinov (University of Illinois at Urbana-Champaign, USA)
Marius Minea ("Politehnica" University of Timisoara, Romania)
Mauro Pezzè (University of Lugano, Switzerland)
Arnd Poetzsch-Heffter (University of Kaiserlautern, Germany)
Andreas Rausch (T.U. Clausthal, Germany)
Natarajan Shankar (SRI, USA)
Yannis Smaragdakis (University of Oregon, USA)
Nigamanth Sridhar (Cleveland State University, USA)
Serdar Tasiran (Koc University, Turkey)

SAVCBS 2008
WORKSHOP INTRODUCTION

This volume contains the proceedings of the *Seventh Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, affiliated with the *Sixteenth ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2008)*. *SAVCBS 2008* took place in Atlanta, Georgia, USA on November 9-10, 2008.

SAVCBS is a venue for discussing how formal (i.e., mathematical) techniques can be or should be used to establish a suitable foundation for the specification and verification of component-based systems. Component-based systems are a growing concern for the software engineering community. Specification and reasoning techniques are urgently needed to permit composition of systems from components. Component-based specification and verification is also vital for scaling advanced verification techniques such as extended static analysis and model checking to the size of real systems. The workshop considers formalization of both functional and non-functional behavior, such as performance or reliability.

SAVCBS aims to bring together researchers and practitioners in the areas of component-based software and formal methods to address the open problems in modular specification and verification of systems composed from components. The workshop seeks to bridge the gap between principles and practice on this research area. The intent of bringing participants together at the workshop is to help form a community-oriented understanding of the relevant research problems and to help steer formal methods research in a direction that will address the problems of component-based systems. For example, researchers in formal methods have only recently begun to study principles of object-oriented software specification and verification, but do not yet have a good handle on how inheritance can be exploited in specification and verification. Other issues are also important in the practice of component-based systems, such as concurrency, mechanization and scalability, performance (time and space), reusability, and understandability. *SAVCBS* aims to provide a venue to brainstorm about these and related topics to understand both the problems involved and how formal techniques may be useful in solving them.

The goals of the workshop are to produce:

1. Contacts and discussion among researchers and practitioners, and
2. A web site that will be maintained after the workshop to act as a central clearinghouse for research in this area.

We enthusiastically thank the authors of submitted papers; their quality contributions and participation are what make a workshop like *SAVCBS* successful. We thank the program committee for their careful reading and reviewing of the submissions. Our PC members have expertise in a wide variety of sub-disciplines related to specification and verification of component-based systems; they include

established research leaders and promising recent Ph.D.s; they come from academia and esteemed research institutes, and hail from all over the world.

We received 15 research paper submissions. All papers were reviewed by 3 PC members, with PC member papers reviewed by 4 PC members. After PC discussions, 6 papers were accepted. As in previous years, we accepted additional submissions as short and poster presentations, reflecting the role of *SAVCBS* to promote discussion and incubation of new ideas for which a full paper may be premature; this year, we accepted 3 papers for short presentations and 4 papers for poster presentations. Two of the accepted poster presentations were withdrawn. Among all of the 15 papers submitted, 2 submissions were rejected.

This year's program also includes a solution to a specification and verification challenge problem based on the "composite pattern". A composite object is one that organizes objects into a tree structure in order to represent a part-whole hierarchy. The point of the pattern is that clients have a uniform interface whether they have a reference to a sub-tree (i.e., a composite object) or a leaf (a single object). The focus of the challenge problem is to specify and verify an invariant that relates each composite node to its children. This invariant is broken when a new child is added, and it remains broken until all the transitive parents of the new node are traversed and adapted. The main challenge is to give a concise specification, especially for the operation that re-establishes the invariant. After the reviewing process, we accepted all 4 submissions to the challenge problem.

This year, we are pleased to have an invited presentation by Wolfgang Emerich of University College London titled "Verification Challenges for Components in Federated Distributed Systems".

Robby (Program Committee Chair)

Jonathan Aldrich (Organizing Committee)
Mike Barnett (Organizing Committee, Challenge Problems Chair)
Dimitra Giannakopoulou (Organizing Committee)
Gary T. Leavens (Organizing Committee)
Natasha Sharygina (Organizing Committee)

SAVCBS 2008 PAPERS

Distributed, Multi-threaded Verification of Java Programs

Perry R. James, Patrice Chalin, Leveda Giannas, George Karabotsos
Dependable Software Research Group
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{perry,chalin,leveda,george}@dsrg.org

Abstract

Extended Static Checking (ESC) is a fully automated formal verification technique and is generally quite efficient, as far as verification tools go, but it is still orders of magnitude slower than simple compilation. Verification in ESC is achieved by translating programs and their specifications into verification conditions (VCs). Proof of a VC establishes the correctness of the program. As can be imagined, proving VCs is computationally expensive: While small classes can be verified in seconds, verifying larger programs of 50 KLOC can take hours. To help address this lack of scalability, we present the multi-threaded version of ESC4 and its distributed prover back-end.

1. Introduction

Extended Static Checking (ESC) [14] is a fully automatic form of static analysis that provides more checks than are available from standard type checking but less than from Full Static Program Verification (FSPV) [11]. It does this by translating source code that has been annotated with specifications to Verification Conditions (VCs), which are boolean expressions in first-order logic. If the VC corresponding to a given method can be discharged then the method is correct with respect to its specification. In ESC, VCs are discharged with the help of Automated Theorem Provers (ATPs).

Technology has progressed incredibly since the first ESC tools were developed. We can formally verify non-trivial applications. While small classes can be verified in seconds, larger programs of 50 KLOC can sometimes take hours to verify. We believe that this is an impediment to widespread adoption of ESC: used to modern incremental software development models, developers have come to expect that the compilation (and ESC) cycles are very quick.

In this paper we describe how ESC4 [11] is able alleviate this problem. ESC4 is the ESC component of JML4 [10], a next-generation research platform that provides an Eclipse-based Integrated Verification Environment (IVE) for JML-annotated Java [18]. It offers a full range of verification techniques, including Runtime Assertion Checking (RAC), ESC, and FSPV. More background information is given in Section 2, and an overview of ESC4 is given in Section 3.

ESC4 [11] is able to verify programs faster than its predecessor, ESC/Java2 [12], and other ESC tools such as Boogie [7]. Our contributions are as follows:

- We take advantage of the modular nature of the verification techniques underlying ESC [19] to analyze the methods in a given compilation unit in parallel. This is possible because the ESC analysis done for a given method is independent of that for any others. (Section 4.1)
- 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 4.2)
- 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 4.3)

While tools exist for verifying distributed and multi-threaded code, 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.

2. Background

ESC4 is a from-scratch rewrite that builds on the lessons learned from earlier projects, principally ESC/Java2. It is part of the JML4 project.

2.1 ESC/Java and ESC/Java2

ESC/Java2 [12] is the successor to the earlier ESC/Java project [14], 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" [14]. The ESC/Java2 project first unified the original program's input language with JML before becoming the platform developed by many research groups.

2.2 JML4

First-generation tools such as ESC/Java and ESC/Java2 are standalone command-line applications that use their own custom Java-compiler front ends to produce an AST. Since the research interest of the maintainers of these tools is JML, and not the underlying Java font end, these tools have not kept up with the latest developments of the Java language.

After much discussion, both within our own research group and with other members of the JML community, it was decided that basing a next-generation JML tooling framework on the Eclipse JDT was the most promising approach.

The result is JML4 [10], a Integrated Verification Environment (IVE) for JML-annotated Java that is built atop the Eclipse Java Development Tooling (JDT).

JML4's first feature set enhanced Eclipse with scanning and parsing of nullity modifiers (nullable and non-null), enforcement of JML's non-null type system (both statically and at runtime) [9] and the ability to read and make use of the extensive JML API library specifications. 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 non-null type system, and
- generating runtime assertion checking (RAC) code.

Since then, work has been underway by several research groups to flesh out JML4 so that it can process all of JML language-level 0 [18].

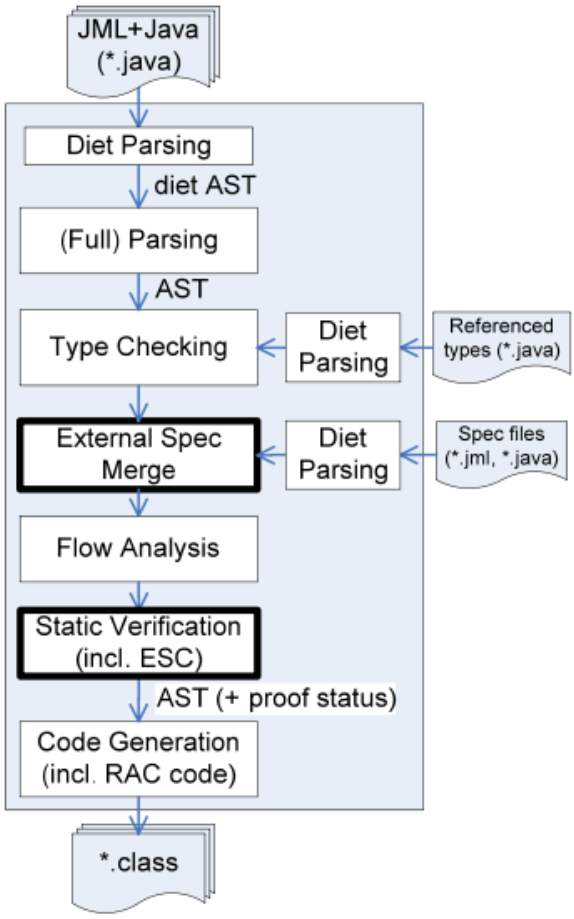


Figure 1. Compiler phases for JML4

The framework has also been enhanced to support static analysis [11], including both ESC and Full Static Program Verification (FSPV). The main compiler phases can be seen in Figure 1.

3. Overview of ESC4

ESC4 [11, 16] is the ESC component of JML4 and is a ground-up rewrite of ESC. Its VC generation is based on Barnett and Leino's innovative and improved approach to a weakest-precondition semantics for ESC [8]. One of the most significant results of this approach is that the size of the VCs produced are linear in the size of the method being analyzed, where earlier approaches generate VCs whose size can be exponential in the worst case.

Figure 2 shows the data flow in ESC4. The fully resolved and analyzed AST produced by the JDT's front end is taken as input. Only those with no front-end-reported errors are processed further by ESC4. The source AST is first converted to a control-flow graph (CFG) as described in [8]. This CFG is similar to Dijkstra's Guarded Command Language [13], except that the guards have been replaced with `assume` statements and the choice operator has

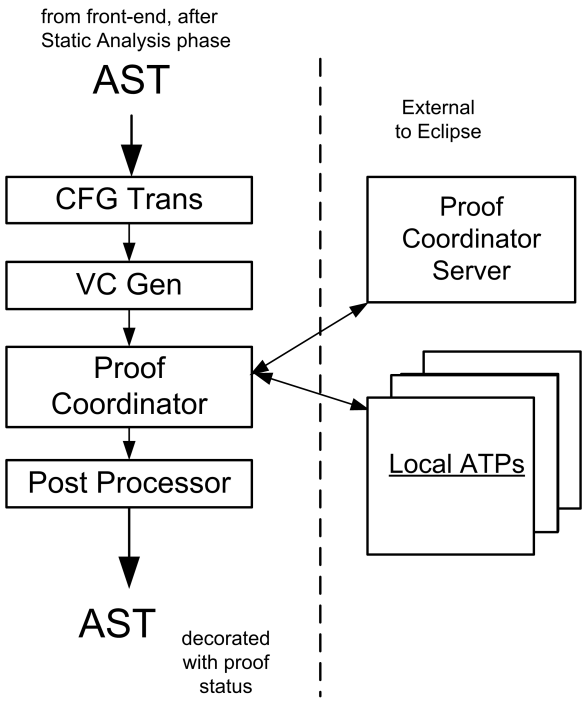


Figure 2. Data flow in ESC4

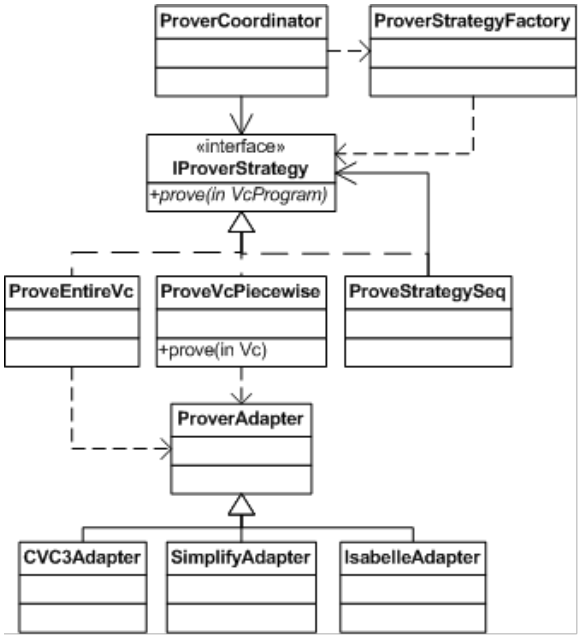


Figure 3. ESC4's prover back-end

been replaced with `goto`s. A VC for each source method is generated from this intermediate form. ESC4's Proof Coordinator is responsible for discharging the VC or reporting why it cannot be discharged. A post-processor reports unprovable VCs to the user through the IVE as failed assertions and attaches the results of the analysis to the original AST. Depending on the compiler options in effect, the code-generation phase may make use of these results to optimize runtime checks.

3.1 Prover back-end

A class diagram for the Prover back-end is shown in Figure 3. A 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 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 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.

`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.

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.

Currently, three ATPs are used: Simplify, CVC3, and Isabelle/HOL. The first two of these are much faster than Isabelle, but Isabelle is able to discharge VCs containing many constructs that the others are not. After trying both Simplify and CVC3 on a sub-VC, we try to prove its negation before resorting to Isabelle. Only after all other attempts fail is Isabelle invoked.

4. Faster ESC

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

4.1 Multi-threading

Using the arguments in Leino's thesis, *Toward Reliable Modular Programs* [19], 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 easy to introduce concurrency.

First-generation tools such as ESC/Java [14] and ESC/Java2 [12] 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 use separate threads to compile individual source files concurrently [3]. 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 3 by default, and 2D VC Cascading can cause those 3 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.

4.2 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, as shown in Figure 4.

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 Figure 5). A Proof 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 3.1 and having it use remote services to discharge the sub-VCs in parallel.

3. **Doubly Remote Prover Coordinator**. Combining the two previous approaches, so that the remote Prover Coordinator itself delegates the responsibility for discharging the sub-VCs to remote services by using the `ProveVcPiecewiseDistributed` strategy, provides yet another alternative. A deployment view can be seen in Figure 5.

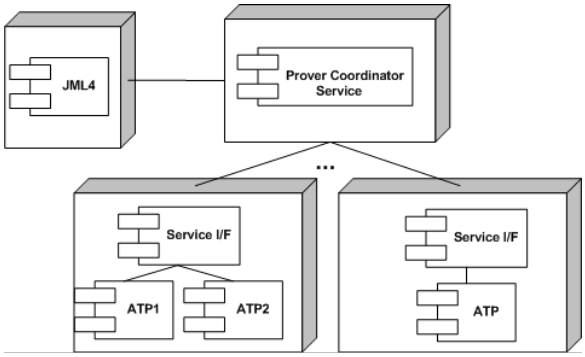


Figure 5. Deployment

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 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.

4.3 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.

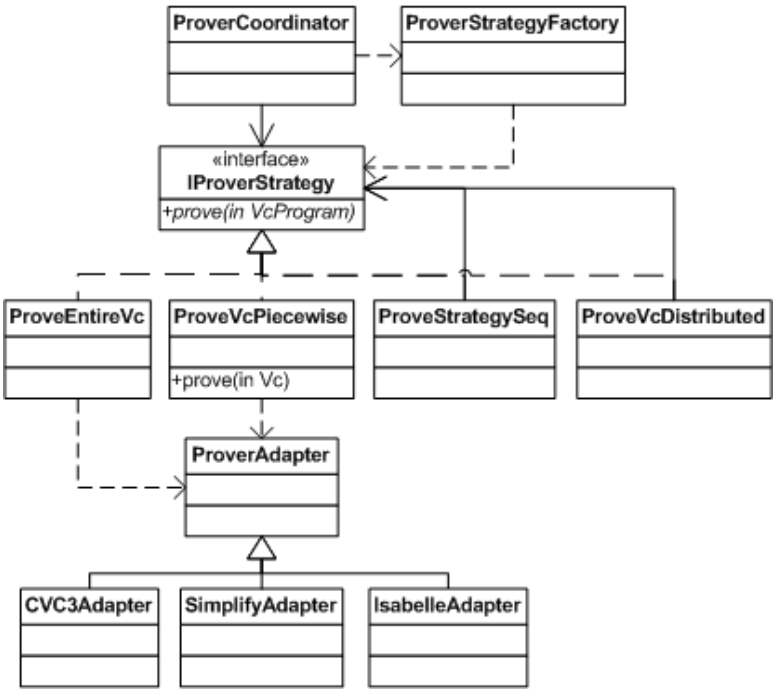


Figure 4. ESC4's distributed back-end

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.

5. 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 1 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 4.2. 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 2 and Figure 6. 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%.

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 [6, 17]. It should have the form

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

Table 1. 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		

Table 2. 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

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.

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. Related Work

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.

6.1 Compilation

As mentioned in Section 4.1, 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 [2]. 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` [4] and `Icecream` [1]. Xoreax sells a product called `IncrediBuild` [5] that coordinates distributed builds from within with Microsoft's VisualStudio.

6.2 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" [20]. 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 [15]. 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.

7. Conclusion

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.

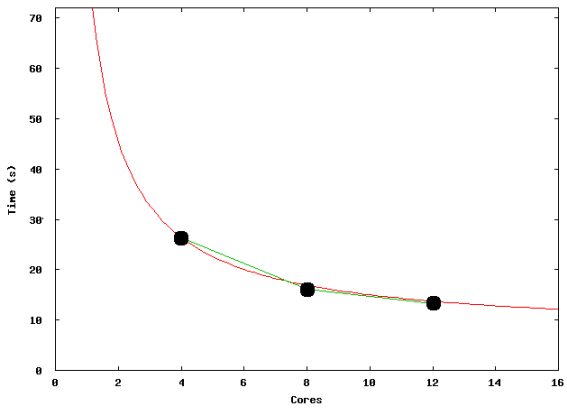


Figure 6. Time (s) vs. Cores

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.

7.1 Next steps

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.

Proof-status caching, as described in [11], would also improve performance during iterative development since only methods that were changed would need to be re-verified.

After making the obvious enhancements, we plan to conduct timing studies 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.

Acknowledgements

We gratefully acknowledge Stuart Thiel's configuration of the servers and other assistance in gathering data for the Validation.

8. References

[1] Icecream - openSUSE, 2006. Homepage at `http://en.opensuse.org/Icecream`.

[2] Parallel - GNU 'make', 2006. Homepage at `http://www.gnu.org/software/automake/manual/make/Parallel.html`.

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

[4] distcc: a fast, free distributed C/C++ compiler, 2008. Homepage at `distcc.org`.

[5] IncrediBuild by Xoreax Software - Distributed Visual Studio Builds, 2008. Homepage at `http://www.xoreax.com/solutions_vs.htm`.

[6] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference* (San Francisco, CA, 1967), pp. 79–81.

[7] BARNETT, M., CHANG, B.-Y. E., DELINE, R., JACOBS, B., AND LEINO, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures* (2006), vol. 4111 of *LNCS*, Springer-Verlag, pp. 364–387.

[8] BARNETT, M., AND LEINO, K. R. M. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, 2005), ACM Press, pp. 82–87.

[9] CHALIN, P., AND JAMES, P. R. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)* (Berlin, Germany, July-August 2007). to appear.

[10] CHALIN, P., JAMES, P. R., AND KARABOTSOS, G. 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* (2007), pp. 47–53.

[11] CHALIN, P., JAMES, P. R., AND KARABOTSOS, G. 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).

[12] COK, D. R., AND KINIRY, J. R. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (2005), vol. 3362/2005 of *LNCS*, Springer Berlin, pp. 108–128.

[13] DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[14] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference* (New York, NY, 2002), ACM Press, pp. 234–245.

[15] HUNTER, C., ROBINSON, P., AND STROOPER, P. Agent-based distributed software verification. In *ACSC '05: Proceedings of the Twenty-eighth Australasian Conference on Computer Science* (Darlinghurst, Australia, 2005), pp. 159–164.

[16] JAMES, P. R., AND CHALIN, P. Enhanced extended static checking in JML4: Benefits of multiple-prover support. In

ACM SAC 2009 (24th Annual ACM Symposium on Applied Computing) (2009).

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

[18] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J. R., AND CHALIN, P. JML reference manual, 2008. Available at `http://www.jmlspecs.org`.

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

[20] VANDEVOORDE, M. T., AND KAPUR, D. 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* (London, UK, 1996), Springer-Verlag, pp. 420–423.

JML and Aspects: The Benefits of Instrumenting JML Features with AspectJ

Henrique Rebêlo
Sérgio Soares
Department of Computing and Systems
University of Pernambuco
Recife, Pernambuco, Brazil
{hemr,sergio}@dsc.upe.br

Ricardo Lima
Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil
{rmfl,phmb}@cin.ufpe.br

Márcio Cornélio
Department of Computing and Systems
University of Pernambuco
Recife, Pernambuco, Brazil
{marcio}@dsc.upe.br

ABSTRACT

The Java Modeling Language (JML) is used to specify designs of Java classes and interfaces. To this end, JML has a rich set of features for specifying methods, including specification inheritance. Thus, the most fundamental motivation for employing JML is to improve functional software correctness of Java applications, and helps to reduce corrective maintenance effort of those applications. Previously, we presented a new JML compiler (ajmlc) that generates aspects (AspectJ) for contract enforcement. This paper describes the main reasons to instrument JML features with AspectJ, with particular emphasis on issues related to instrumentation code size — we also defined guidelines to use ajmlc that always generate compact instrumented code than the classical JML compiler (jmlc). In addition, we discuss the analogy between JML and AspectJ, and how the ajmlc also deals with Java ME applications, which is not possible with jmlc. Moreover, we implemented other JML features such as the new the new assertion semantics based on "strong validity" presented elsewhere. The paper includes studies to compare the final code generated by ajmlc with the one produced by jmlc. Results indicate that the overhead in code size produced by our compiler is very small when using the proposed guidelines, which is essential for Java ME applications.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Languages Classifications—*JML*

General Terms

Languages, Experimentation

Keywords

Design by contract, JML language, JML compiler, JML new assertion semantics, aspect-oriented programming, AspectJ, AspectJ weaving

1. INTRODUCTION

The Java Modeling Language (JML) [19, 18] is a formal behavioral interface specification language for Java. The JML compiler (jmlc) reads a Java program annotated with JML specification and produces instrumented bytecodes. Such additional code checks the correctness of the program against restrictions imposed by the JML specifications.

In a previous work [24], we proposed a new JML compiler known as ajmlc (AspectJ JML compiler). The ajmlc employs AspectJ [14, 15] (generative programming) to implement JML features (specifications) without generics. The AspectJ compiler translates such features into an instrumented bytecode, which performs a runtime checking of those features. The ajmlc generates code compliant with both Java SE and Java ME [23] applications. Similarly, Jose [10] is a tool that uses AspectJ to instrument Java programs. Jose tool adopts a different specification language to specify Java programs. Thus, its semantics is different from that of JML. However, the instrumentation provided by the Jose tool is not complaint with Java ME applications, whereas ajmlc does. Moreover, the language JCML [9] is a subset of the JML language targeting Java Card applications. Different from ajmlc, the JCML compiler does not use Aspect Oriented Programming. As with the original JML compiler, it instruments JML features using standard Java code with a few restrictions imposed by the Java Card platform.

Based on our previous results and the new results addressed by this paper, we try to answer properly the following research questions:

- Does AOP represent the JML features (specifications) conveniently?
- When is it beneficial to aspectize JML features in relation to both source and bytecode instrumentation? When it is not?
- How to check JML features during runtime?
- How to modify properly the code generation of the JML compiler for generating aspects that support runtime assertion checking of JML features in Java ME applications (in a constrained environment)?
- What is the relationship between the generated aspects?
  What is the order of aspects generation to provide an effect like the *wrapper approach* present in the classical JML compiler (jmlc)?

The main contributions of this paper are: (1) answering the above research questions throughout the paper; (2) describing the analogy between JML and aspects (AspectJ); (3) extending ajmlc to support the new assertion semantics; (4) generating instrumented bytecode to verify constrained methods during class initialization (this feature is not supported by jmlc); (5) generating instrumented bytecode when necessary (this feature is also not supported by jmlc); (6) Conducting study between ajmlc with two different AspectJ weavers and the original JML compiler to investigate the overhead in the code size.

This paper is organized as follows. The next section 2 presents the background of Java Modeling Language (JML). Section 3 describes the ajmlc compiler as well as its new issues addressed. Some results of conducted studies between our and original JML compiler are discussed in Section 4. Section 5 discusses related work. The last section contains the conclusions and points directions for future work.

2. JML: BACKGROUND

The Java Modeling Language (JML) [19, 18] is a specification language to describe the expected behavior of Java modules — Java modules are classes and interfaces. It combines the Design By Contract (DBC) approach [21] of Eiffel [22] and the model-based specification approach of the Larch family [12] of interface specification languages, with some elements of the refinement calculus. Hence, JML specifications contains pre-, postconditions, and invariant predicates based on Hoare-style [13].

Java comments beginning with the symbol @, which are interpreted as JML annotations (see example in Figure 1). One can use JML to specify the behavior of types (type specifications) or methods (method specifications). An `invariant` clause is a type specification. For instance, the predicate `I` denotes an invariant condition that must be true after the execution of all constructors of the class `Foo`. Moreover, this predicate is supposed to be true before and after the execution of all methods of the class `Foo`. On the other hand, `requires`, `ensures`, and `signals` clauses represent method specifications. `Requires` specify the method precondition, whereas `ensures` and `signals` are respectively used to define the normal and exceptional postconditions of the method `foo`.

```
public class Foo {
  //@ invariant I;
  /*@ requires P;
    @ ensures Q;
    @ signals (FooException e) R(e);
    @*/
   public void foo() throws FooException
   {...}
}
```

Figure 1: Example of JML specification.

2.1 JML assertion semantics

The current JML compiler implements the assertion semantics based on "strong validity" proposed by Chalin's work [25]. Thus, instead of using the classical two-valued logic in the older approach proposed by Cheon's work [4], the JML compiler uses now a three-valued logic semantics. In this way all logical operators behave similarly in both Java and JML. Using the new semantics, an assertion can be satisfied (true), violated (false) or invalid (when evaluation does not complete successfully). More details about the new assertion semantics, refer to [25].

2.2 The JML compiler

The JML compiler (*jmlc*) [3] was developed at Iowa State University. It is a runtime assertion checking compiler that converts JML annotations into automatic runtime checks.

Design

Jmlc is built on top of the MultiJava compiler [6]. It reuses the front-end of existenting JML tools [2] to verify the syntax and semantics of the JML annotations and produces a typechecked abstract syntax tree (AST). The compiler introduces two new compilation passes: the "runtime assertion checker (RAC) code generation"; and the "runtime assertion checker (RAC) code printing". The former modifies the AST to add nodes for the generated checking code; the latter writes the new AST to a temporary Java source file.

For each Java method three *assertion methods* are generated into a temporary Java source file: one for precondition checking, and two for postcondition checking (for normal and exceptional termination). They are invoked before method call (precondition checking), after method call (normal postcondition checking) and when an exception is thrown by the called method (exceptional termination checking). Finally, instrumented bytecode is produced by compiling the temporary Java source file through the MultiJava compiler. The instrumented bytecode produced contains *assertion methods* code embedded to check JML contracts at runtime.

Wrapper approach

The *wrapper approach* [3, 4.1.3] is a strategy used by the JML compiler to implement the assertion checking. Each method is redeclared as private with a new name. Then, a method known as *wrapper method* is generated with the name of the original method. Its surrounds the original method (now with a new name) with the assertion methods. Hence, client method calls the wrapper method, which is responsible for calling the original method with appropriate assertion checks (e.g., precondition checking). The JML compiler is responsible for controlling the order of execution of assertion methods.

Figure 2 depicts the wrapper approach strategy. If a client calls the original method, the call goes to the wrapper method. In this way, the precondition assertion method is the first assertion method called, and then only if the precondition is satisfied, it calls the original method. After calling the original method, if it terminates normally, the normal postcondition assertion method is called; otherwise, the exceptional postcondition assertion method will be called.

3. AJMLC: A JML COMPILER TARGETING ASPECTJ CODE

In this section we present the analogy of JML and AspectJ aspects. We explain the reason to aspectize JML features. The remaining reasons only will be understood in the Sec-

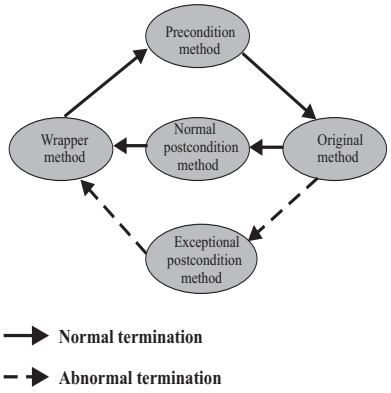


Figure 2: Wrapper approach strategy.

tion 4 with the studies results. Furthermore, we concentrate on new assertion semantics verification recently provided by the ajmlc, the reason that ajmlc can be used with Java ME applications, and among other issues. Some implementation details of our compiler will also be considered. A detailed implementation mechanism has already been available in a previous work [24].

3.1 AspectJ Overview

AspectJ [14, 15] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of units of a program which affect (crosscut) other concerns. Such units are called crosscutting concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either scattering or tangling code, or both. Thus, this separation of concerns allows better modularity, avoiding tangled code and code spread over several units. Consequently, the system maintainability is also increased. Programming with AspectJ explores both objects and aspects concepts to separate concerns. Object-oriented programming can be used when the concern are well modeled as objects. If it is not the case, concerns that crosscut the objects are separated using units called aspects, and those are composed with the objects of a system by a process called weaving. By weaving AspectJ aspects with standard Java code, we obtain a new AspectJ application.

The main construct of the AspectJ [14, 15] language is called *aspect*. Each aspect defines a functionality that crosscuts others (crosscutting concerns) in a system. An aspect can declare attributes and methods, and can extend another aspect by defining concrete behavior for some abstract declarations. An aspect can affect both static and dynamic structure of Java programs. The static structure might be changed by introducing new methods and fields to an existing class, as well as converting checked exceptions into unchecked exceptions, and changing the class hierarchy. The dynamic structure is changed by intercepting specific points, called join points, of the program execution flow and adding behavior before, after, or around the join point.

3.2 Ajmlc design

Similarly to Cheon [3], we reuse the front-end of the JML compiler, known as JML Type Checker [2]. Then, we modify the code generation part of the original JML compiler[1] to introduce other two new compilation passes: the *Aspect RAC* code generation; and the *Aspect RAC* code printing. The former produces assertion checker code from the typechecked AST, whereas the latter writes the assertion checker code to a temporary Aspect source file. We traverse the typechecked AST generating *Aspect Assertion Methods* (AAM) for each Java method in a temporary Aspect source file: one for precondition checking, and another for both kinds of postconditions in JML (normal and exceptional). Eventually (when necessary) we also generate AAM for both kinds of invariants in JML (instance and static). These AAM are compiled through the AspectJ compiler (ajc or abc [1]), which weaves the AAM with the Java code. The result, unlike jmlc, is an instrumented bytecode compliant to both Java SE and Java ME applications.

3.3 Ajmlc runtime environment

The instrumented bytecode produced by the ajmlc contains not only its normal content (usually generated by javac), but also has embedded code (assertion methods) to checks JML's features during runtime.

In order to run and check those assertion methods of the bytecode generated by ajmlc, we use part of the AspectJ runtime environment library (answer to research question 3 discussed in Section 1). We need only part of the AspectJ library, because only a few AspectJ constructs must be used during runtime. This is due to compatibility needed by the instrumented code to deal with Java ME applications. For more information about ajmlc with Java ME applications and its required AspectJ library refer to section 3.7 and 4.

3.4 The analogy between JML and Aspects

As pointed out by Filman and Friedman [11], the Figure 3, is an example of quantification. Aspect-oriented programming languages such as AspectJ [14, 15] allow programmers to define quantified programmatic assertions.

To this end, AspectJ provides property-based crosscutting to affect from small to a large number of Java modules (e.g., classes, interfaces, and methods). To perform such property-based crosscutting, by using AspectJ, one can use a feature known as wildcarding ($*$) in pointcut designators. Consider the following example:

```
execution(* T.*(..))
```

This AspectJ construct identifies executions to any method (with any return and any parameters type) defined on type `T`.

The invariants analogy

The behavior of quantification can be addressed similarly by using JML invariants. For example the JML instance invariants must be satisfied by all instance methods of the current type and also subtypes (quantification). In Figure 3, we have a behavior of instance invariant checking by using aspects (note the use of wildcards). On the other hand, if we use pure JML, the following clause replaces both `before` and `after` AspectJ advices [14, 15] depicted in Figure 3.

```
//@ instance invariant i == 10;
```

[1]Part of the code of the original JML compiler that we used to implement the ajmlc was based on the JML 5.5 version available to download at `http://sourceforge.net/projects/jmlspecs`.

As mentioned before, this JML clause defines a quantification property (`i == 10`) that must hold by all instance methods in type `T` and also in its subtypes (see Figure 3).

Behavioral subtyping analogy

Regarding specification inheritance in JML, instance methods with specification cases (e.g., pre- and postconditions) must be satisfied by the current type and also subtypes. For example, suppose a scenario with a JML precondition declared in method `m` of type `T` and we have an subtype of `T` (that extends it) called `S` that overrides the method `m` with other specification cases. Thus, if we have an object of type `S`, we must satisfy the current specification cases of method `m` in combination with the inherited ones (from type `T`), resulting in disjunction for preconditions and conjunction for postconditions [17]. Thus, we can also specify this behavior know as behavioral subtyping using aspects — aspects that checks conditions (assertions) defined in the specification cases locally by the method `m` in combination with the inherited conditions (from the type `T`).

Other analogies

We discussed two points in JML and AspectJ that their behavior work in the same way. We also showed and argued that those points identified in JML are in fact quantification points that can be implemented using AspectJ. However, there are other quantification points in JML that certainly can be expressed with AspectJ. Examples of such quantification points [19] that can be found in JML not limited to:

- instance and static constraint specifications (is a JML type specifications like invariants);
- refinement;
- model-programs;
- non-functional properties;
- so forth.

AspectJ and JML a perfect match

Based on the above argumentations, we showed that JML has properties that cutting across several modules — the concern know as contract enforcement present in JML which is classified as a crosscutting concern [20, 7]. Since AspectJ provide means to deal with crosscutting properties, we conclude that AspectJ can implement properly various JML features (answer to research question 1 discussed in Section 1).

Feldman's work [10] provides another evidence that AspectJ can be used to implement contract enforcement concern [20, 7]. Section 5 presents the main points related to such a work.

3.5 Expression evaluation with new assertion semantics

Ajmlc was restructured to deal with the new assertion semantics proposed by Chalin's work [25] and implemented by the current JML compiler. Considering this semantics, a clause can be entirety executable or not. In this way, we generate into aspects two try-catch blocks:

- one to handle non-executable exceptions discovered at runtime;

```
public class T {
    int i = 10;

   public void m() {...}
   public void n() {...}
   public void o() {...}
}

privileged Aspect_T{
  before (T current) :
    execution(!static * T.*(..)) &&
    within(T+) &&
    this(current){
      if (!(current.i == 10)) {
        throw new RuntimeException("");
      }
    }

  after (T current) :
    execution(!static * T.*(..)) &&
    within(T+) &&
    this(current){
      if (!(current.i == 10)) {
        throw new RuntimeException("");
      }
    }
}
```

Figure 3: Example of AspectJ quantification.

```
public class T{
  public int x, y;
  //@ requires b && x < y;
  public void m(boolean b)
  { ... }
}
```

Figure 4: Simple method with a precondition.

- another to handle all other exceptions, such as *NullPointerException* raised during assertion checking by a method.

In order to see an example of this approach, consider a method `m` declared in a type `T` with a simple precondition (see Figure 4).

An AspectJ `before advice` [14] is generated by the ajmlc to instrument such a precondition. This `before` advice contains the above mentioned two try-catch blocks. In Figure 5, we can observe the resulting instrumentation code generated by ajmlc. Note that the presence of the `JMLEvaluationError`, which is a new JML assertion error [25] responsible for handling invalid assertion evaluations.

3.6 Ordering of advice executions into an aspect

One AspectJ aspect can have several advices (e.g., `before`) to apply to a particular named or anonymous pointcut. As the advices are declared into the same aspect, we should take into account their order declaration. In this way, the advice that appears first lexically inside the aspect executes

```
public boolean T.checkPre$m$T(boolean b){
  return ((b) && (x < y));
}

before (T current, boolean b) :
   execution(void T.m(boolean)) &&
   within(T) &&
   this(current) && args(b) {
     boolean rac$b = true;
     try {
       rac$b = current.checkPre$m$T(b);
       if(!rac$b){
         throw new
         JMLInternalPreconditionError("");
       }
     } catch (JMLNonExecutableException
       rac$nonExec) {
         rac$b = true;
     } catch (Throwable rac$cause) {
         if(rac$cause instanceof
            JMLInternalPreconditionError) {
           throw (JMLInternalPreconditionError)
                 rac$cause;
         }
         else {
           throw new JMLEvaluationError("");
         }
     }
   }
```

Figure 5: Evaluation of precondition in the new assertion semantics.

first. "The only way to control precedence between multiple advice in an aspect is to arrange them lexically [16]." Thus, ajmlc generate AspectJ advices carefully in order to respect the JML semantics (answer to research question 5 discussed in Section 1). The order of generation is as follow:

1. generate a `before` advice to check static invariants;
2. generate a `before` advice to check instance invariants;
3. generate `before` advice to check preconditions of existing methods (including constructors);
4. generate `after returning` advices and `after throwing` advices or `around` advices (if we have old expressions) to check postconditions (normal and exceptional postconditions) of existing methods (including constructors);
5. generate a `after returning` and a `after throwing` advices to check instance invariants;
6. generate a `after returning` and a `after throwing` advices to check static invariants;

The above ordering to generate AspectJ code is extremely important to keep the classical ordering of contract checking posed by JML semantics. For example, a method to be executed must obey some conditions in a certain order:

1. check invariants (static and instance invariants) before method execution;
2. check preconditions before method execution;
3. check postconditions after method execution (normal postconditions when the method terminates normally and exceptional postconditions when the method terminates abnormally);
4. check invariants (static and instance invariants) after method execution.

These ordering is respected by generated aspects to check JML features during runtime — such aspects ordering have an analogy with the Cheon's *wrapper appraoch* [4], because they have the same effect during runtime checking (ordering to call the assertion methods, such as precondition checking method).

3.7 Ajmlc and Java ME applications

The main benefit in using ajmlc is that one can specify and verify during runtime Java ME applications [23] with JML. To this end our compiler only generates aspects that avoids AspectJ constructs that are not supported by Java ME, such as `cflow` pointcut [14, 16] (answer to research question 4 discussed in Section 1).

3.8 Ajmlc optimizations

Concerning Java ME applications, we introduced several optimizations in ajmlc in order to generate small instrumentation code as much as possible due to constrained environments like Java ME platform.

Compiling empty classes

The jmlc compiler assumes a standard configuration for classes. Thus, even if one defines an empty class, basic instrumentation is generated [19, 18] for:

1. class verification
   - Static and non-static invariant/constraint checking;
   - Static and non-static constraint pre-state expressions checking.
2. default constructor verification
   - Assertion checking wrapper;
   - Precondition checking;
   - Normal postcondition checking;
   - Exceptional postcondition checking.
3. other methods (e.g., for dynamic calls using reflection)

In this way, the jmlc compiler generates 11.0 KB (source code instrumentation) and 5.93 KB (bytecode instrumentation) even for a empty class like:

```
public class Empty { }
```

In contrast to jmlc compiler, our compiler does not generate code for empty classes.

Code instrumentation

Code size is an important issue for Java ME applications. Our compiler avoids code generation as much as possible. Table 1 compares the jmlc and ajmlc compilers when no specification is provided.

Limitation of the jmlc compiler solved by the ajmlc compiler

The current implementation of the jmlc compiler has one limitation:

JML clauses	jmlc generates	ajmlc generates
requires	yes	no
ensures	yes	no
signals	yes	no
invariant	yes	no

Table 1: Difference between jmlc and ajmlc during the generation code.

```
public static x;
//@ static invariant x > 0;

public static void m() { x = -3; }

static{
  m();
}
```

Figure 6: Example of non checked type invariant when it is called.

1. When constrained methods are called into static blocks during the class initialization, jmlc does not check the constrains and the method is always executed even if the condition is `false`.

Figure 6 shows an example where the method `m` is constrained with the invariant (`x > 0`) and a call (`m()`) that violates the invariant is made inside a static block. As a result, no assertion violation is raised. Cheon's compiler [3] does not generate instrumented bytecode properly to deal with this limitation. However, the ajmlc always verifies constrained methods when called into static blocks. This benefit is automatically gained just by using aspects to instrument the JML features.

4. STUDY

In our previous work [24], we evaluated our compiler (ajmlc) by using a Java ME application. This was fundamental to investigate our proposed approach in a Java ME environment. In this section we evaluate our compiler employing three Java applications. Such applications was extracted from the JML literature, as described bellow.

Scenario

We have compiled three Java programs annotated with JML using both ajmlc (our compiler), and the jmlc compiler (Cheon's compiler [3]). Such programs are described in three works: (1) the hierarchy classes `Animal`, `Person`, and `Patient` [17]; (2) the class `IntMathOps` [19], and (3) the class `StackAsArray` [3]. Moreover, we have used our ajmlc with two different weaving processes: using the standard AspecJ compiler (ajc) [14]; and the abc compiler [1], which is a complete implementation of AspectJ with some optimizations.

As a important point for our study, we removed the JML specifications from the class `Person`. This choice is to show that our compile only generate instrumented code when necessary.

Results

Considering the scenario described above, Table 2, Table 3 and Table 4 present the results of the compilation size that we obtained by using both compilers. As can be seen, we analyzed instrumented source code size, instrumented bytecode size, and Jar size all in kbytes (KB). Considering our compiler (ajmlc), we used the same AspectJ aspect code (source) generated for both weaving processes (using ajc, and abc compilers). We observed that the ajmlc compiler using the ajc weaver introduces a big overhead in the instrumented bytecode size (see Table 3), whereas in relation to instrumented source code size, ajmlc generates a smaller code (see Table 2). On the other hand, our approach produces a far smaller instrumented bytecode and source code when the abc weaver is employed (see Tables 3 and 2). Concerning Jar size we observed that the final deployed applications for both ajmlc with ajc and abc weavers are smaller to ones related to the jmlc. This happens because the lib Jar size necessary to evaluate assertions during runtime for our compiler is smaller than the lib Jar size for jmlc. It is important to note that we take into account only the JML features available by our compiler. Thus, we removed the from the jmlc runtime library the part that is nor supported yet by our compiler — this gives a more fair comparison. Therefore, the user is free to choose the AspectJ weaver. However, based on the results, we recommend the usage of ajmlc with the abc weaver (for most cases). This choice is particulary important for Java ME applications.

Guidelines

Based on the results presented in Tables 2, 3, and 4, we briefly present steps to use our ajmlc compiler (answer to research question 2 discussed in Section 1). These steps are the guidelines for its usage:

1. If the application is not compiled in its entirety by the JML compiler — if at least $\approx 33\%$ of the application is free of the JML instrumentation effect (as occurs with the Hierarchy application showed above), we recommend to use ajmlc with both ajc or abc weavers. Even if the application is a Java ME application. But, be aware that by always using abc we got better results;

2. If the application is compiled in its entirety by the JML compiler — we recommend only in the case of Java ME applications to use ajmlc with abc weaver;

3. If the user always need to take maximum of the AspectJ optmization — we alwys recommend to use ajmlc with abc weaver.

These guidelines is to provide a way to choose the best AspectJ weaver to use. Although, our compiler always need smaller memory space during deploying (because the discrepancy of the size of the runtime libraries from ajmlc and jmlc).

5. RELATED WORK

JMLC (Java Card Modeling Language) [9] is a is a subset of the JML language. The JCML compiler (jcmlc) generates bytecode compliant with Java Card applications. However, its instrumentation does not employ AspectJ to implement the JML contracts. The jcmlc translates only JML lightweight specifications, whereas our compiler handles both lightweight and heavyweight specifications. The jcmlc does not support inheritance of specifications, which our compiler

Table 2: Instrumentation source code size results

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	28.8	4.8	4.8
Person	27.4	0.5	0.5
Patient	26.2	9.6	9.6
IntMathOps	18.2	2.0	2.0
StackAsArray	55.7	9.2	9.2

Table 3: Instrumentation bytecode size results

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	13.3	17.0	5.5
Person	**11.7**	**2.3**	**0.7**
Patient	12.7	25.3	7.4
IntMathOps	9.39	5.4	2.3
StackAsArray	21.7	23.2	6.2

does. On the other hand, the jcmlc handles quantifiers such as `forall`, which are not treated by our compiler.

Feldman *et al.* [10] presents a DBC tool for Java, known as *Jose*. This tool adopts a private DBC language for expressing contracts. Similar to our approach, Jose adopts AspectJ for implementing contracts. The semantics of postconditions and invariants in Jose are distinct from JML. Jose states that postconditions are simply conjoined without taking into account the corresponding preconditions. Moreover, it establishes that private methods can modify invariant assertions. In the JML semantics, if a private method violates an invariant, an exception must be thrown. Unlike our compiler, Jose generates bytecode not compliant with Java ME.

Pipa [26] is a behavioral interface specification language (BISL) tailored to AspectJ. It uses the same approach (based on annotations) of JML language to specify AspectJ classes and interfaces, and extends JML with a few new constructs in order to specify AspectJ programs. The Pipa language also supports aspect specification inheritance and crosscutting. Pipa specifies AspectJ programs with pre-, postconditions, and invariants. Moreover, Pipa also can specify aspect invariants and the "decision" whether or not to call the proceed method within the around advice (using the proceed extended annotation). The aim in designing Pipa based on JML is to reuse the existing JML-based tools. In order to make this possible the authors developed a tool (compiler) to automatically transform an AspectJ program with Pipa specifications into a standard Java program with JML specifications. To this end, the authors modified the AspectJ compiler (ajc) to retain the comments during the weaving process. After the weaving process, all JML-based tools can be applied to AspectJ programs. Therefore, the main goal of Pipa is to facilitate the use of JML language to verify AspectJ programs. On the other hand, we use AspectJ to implement JML features and verify Java programs.

6. CONCLUDING REMARKS

In this paper we discussed the benefits to use AOP to instrument JML features. We also discussed the analogy between JML and AspectJ that justify the use of AspectJ to instrument JML features (treated as crosscutting concern). In this way the issues covered throughout the paper provide means to answer the research questions pointed out in Section 1.

Table 4: Jar size results

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
hierarchy classes	33.6	18.7	10.7
IntMathOps	20.6	7.5	4.7
StackAsArray	25.2	11.7	6.6

Another major contribution of this paper is that, unlike jmlc, our compiler (ajmlc) generates instrumented bytecode to verify constrained methods within static blocks during the class initialization. We also present three examples of Java programs annotated with JML to investigate the overhead in code size produced by two different AspectJ weavers. Such results provide an evidence that our approach generates smaller code than the original JML compiler when using the abc AspectJ weaver. Moreover, such results showed that in relation to application Jar sizes, ajmlc with either ajc and abc produces a smaller application size than jmlc. These results are essential when considering Java ME applications. We also presented some guidelines useful to choose properly which AspectJ weaver to employ.

We believe that the usage of aspects to implement a JML compiler introduces a new level of modularity. In other words, our approach is not invasive (the Java source code is not tangled and scattered with the generated assertion methods to check JML features during runtime). This gives more flexibility to extend the compiler with other JML constructs and to optimize the current implementation (since our source code instrumentation is less complexity resulting in a smaller source code instrumentation). In addition, optimizations in the weaven process are automatically inherited by our compiler when using abc.

As a future work, we also plan to address a problem suggested by Cheon [3]: to support assertion checking in a concurrent environment (e.g., multi-threaded program). We also intend to conduct more experiments using weavers that implement optimization techniques for AspectJ, including the work by Cordeiro [8]. Such a work provide some optimizations in the AspectJ abc compiler, which can improve the instrumented code generated by the ajmlc. As another future work, we intend to perform quantitative studies to compare the instrumented code generated by the classical jmlc compiler and the ajmlc compiler. These quantitative studies will respect to important software engineering attributes [5], such as composability, coupling, cohesion, number of attributes and operations. Finally, in addition to quantitative studies and code size conducted in this paper, a performance comparison would also be an interesting future work to investigate, especially in the Java ME context.

7. ACKNOWLEDGMENTS

We would like to thank Professor Gary Leavens for his comments and stimulating discussions on earlier topics of this paper. We would also like to thank Professor Patrice Chalin, and Perry James for their several and helpful dis-

cussions about JML and its semantics. Special thanks to Fernando Calheiros, he shared a substantial amount of domain knowledge related to Java ME with AspectJ.

This work was partially supported by CAPES and FINEP, brazilian research agencies.

8. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[2] L. Burdy et al. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[3] Y. Cheon. *A runtime assertion checker for the Java Modeling Language*. Technical report 03-09, Iowa State University, Department of Computer Science, Ames, IA, April 2003. The author's Ph.D. dissertation.

[4] Y. Cheon and G. T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 149–158, New York, NY, USA, 2005. ACM.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[6] C. Clifton et al. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[7] C. Constantinides and T. Skotiniotis. Reasoning about a Classification of Cross-cutting Concerns in Object-Oriented Systems. In *Second Workshop on Aspect-Oriented Software Development (Workshop Aspektorientierte Softwareentwicklung der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung)*, Bonn, Germany, February 21-22, 2002.

[8] E. Cordeiro et al. Optimized compilation of around advice for aspect oriented programs. *Journal of Universal Computer Science*, 13(6):753–766, 2007.

[9] U. Costa et al. Specification and Runtime Verification of Java Card Programs. In *Brazilian Symposium on Formal Methods (SBMF)*, Oct. 2008.

[10] Y. A. Feldman et al. Jose: Aspects for design by contract80-89. *sefm*, 0:80–89, 2006.

[11] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, 2000.

[12] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

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

[14] G. Kiczales et al. Getting Started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[17] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Z. Liu and H. Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260, pages 2–34, Nov. 2006.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[19] G. T. Leavens et al. Jml reference manual. Department of Computer Science, Iowa State University. Available from url http://www.jmlspecs.org, Apr. 2007.

[20] M. Marin et al. A classification of crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA, 2005. IEEE Computer Society.

[21] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[22] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[23] V. Piroumian. *Wireless J2me Platform Programming*. Prentice Hall Professional Technical Reference, 2002. Foreword By-Mike Clary and Foreword By-Bill Joy.

[24] H. Rebêlo et al. Implementing Java Modeling Language Contracts with AspectJ. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA, 2008. ACM.

[25] F. Rioux and P. Chalin. Effective and Efficient Runtime Assertion Checking for JML Through Strong Validity. In *Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs (FTfJP'07)*, 2007.

[26] J. Zhao and M. C. Rinard. Pipa: A behavioral interface specification language for aspectj. In *Proc. Fundamental Approaches to Software Engineering (FASE'2003) of ETAPS'2003*, Lecture Notes in Computer Science, Apr. 2003.

Total Correctness of Recursive Functions using JML4 FSPV

George Karabotsos, Patrice Chalin, Perry R. James, Leveda Giannas
Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{g_karab,chalin,perry,leveda}@dsrg.org

ABSTRACT
JML4 is a next generation tooling and research platform for JML. JML4, currently in development, aims to support the integrated capabilities of Runtime Assertion Checking (RAC), Extended Static Checking (ESC), and Full Static Program Verification (FSPV). In this paper, we present the JML4 FSPV Theory Generator (TG) that aims to study the adequacy of Isabelle/Simpl as the underlying verification condition language. In particular we study Isabelle/Simpl with respect to proving total correctness of recursive programs. Simpl is a Hoare-based logic for a sequential imperative programming language along with a verification system. It is written in Isabelle/HOL and has been proven sound and relative complete.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract, Correctness proofs*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—Mechanical verification.

General Terms
Reliability, Languages, Theory, Verification.

Keywords
Java, Java Modeling Language, Full Static Program Verification.

1. INTRODUCTION
The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) for Java [14]. A number of tools exist that recognize JML annotated Java programs and can help in demonstrating their correctness [4]. These tools perform verification using one or more of three main verification methods: Runtime Assertion Checking (RAC) [7], Extended Static Checking (ESC) [8], and Full Static Program Verification (FSPV) [12].

While RAC and ESC are fully automated and generally easy to use, these verification techniques are either unsound and/or incomplete by nature of the technique. Unfortunately, this is unacceptable for safety and security critical applications (e.g. SmartCard applications such as electronic purses used in commercial transactions and medicare cards used to hold vital patient information) for which soundness and completeness are vital. FSPV, on the other hand, has the potential to be both sound and complete. In this paper, we present the *FSPV Theory Generator (TG)*, the FSPV component of JML4—a next generation tooling and research platform for JML. In particular, we present initial results with respect to proving the total correctness of recursive functions. To our knowledge, the JML4 FSPV TG is the *first*:

- JML tool to enable the total correctness of recursive functions to be proven, such as the one shown for Factorial in Figure 1, and
- FSPV tool to be based on an underlying theory that has been proven sound and complete, and this within a mechanical theorem prover.

Creation of the FSPV TG is also timely, since neither of the two "first generation" FSPV tools (JACK, LOOP) is still being actively maintained.

We present:

- The *translation* process used to generate Isabelle/Simpl [20] theories from Java programs.
- Our *experience* in generating and proving Simpl theory Verification Condition (VC) lemmas for JML annotated Java programs.

Isabelle/Simpl is a theory built atop Isabelle/HOL for an IMP-like [22] sequential imperative programming language with loops and procedures supported by specification constructs (e.g., via pre- and post-conditions).

The rest of the paper is structured as follows. In the next section, we describe Isabelle, Simpl, and JML4. Section 3 presents the FSPV TG followed by an account of its use and subsequent verification of its generated theories in Section 4. In Section 5 we present related work. Finally conclusions and future work are given in Section 6.

2. BACKGROUND

2.1 Isabelle
Isabelle [18] is a theorem proving framework. It provides the necessary proving apparatus to define new logics. This machinery includes Isabelle's meta-logic (Isabelle/Pure), the classical reasoner, and the simplifier. Additionally, existing logics can be extended, thus defining new ones. Newly constructed object logics can be further enhanced with new syntax by making use of Isabelle's syntax transformations. These transformations can be specified using relatively simple rules defined within the theory or

```
public class Factorial {
  //@ requires n >= 0;
  //@ ensures  \result ==
  //@   (\product int j; 1 <= j && j <= n ; j);
  //@ measured_by n;
  public static int fac(final int n) {
    if(n == 0)
      return 1;
    else
      return n * fac(n-1);
  }
}
```

Figure 1: Recursive factorial method

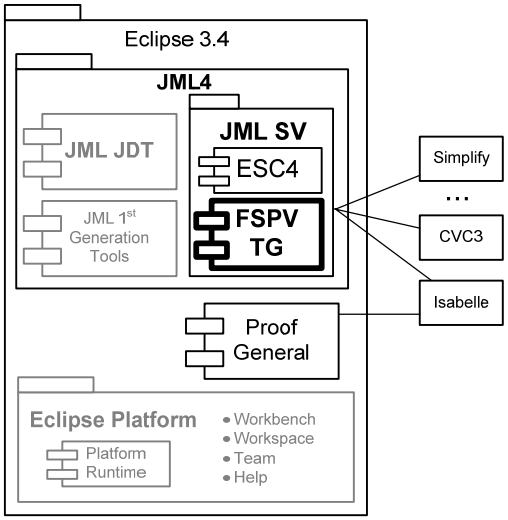


Figure 2: JML4 component diagram

with more complex but more powerful translation functions coded in ML.

Isabelle/HOL, a realization of High Order Logic for Isabelle, is just one of these logics defined atop of Isabelle/Pure. It is the most complete of all of the object logics written for Isabelle so far. This reason, among others, is why Isabelle/HOL has served as the basis for a number of additional logics. Some of these include the Logic for Computable Functions (Isabelle/HOLCF), and logics for sequential imperative programs with Hoare semantics defined such as Bali [17] and Simpl.

2.2 Simpl

Simpl [7] is a theory written and *proven* sound and complete in Isabelle/HOL for a generic sequential imperative programming language. The Simpl theory includes definitions of syntax, big- and small-step operational semantics, a set of Hoare rules both for partial and total correctness, and weakest-precondition semantics (via the `vcg` and `vcg-step` proof methods) [9]. It is expressive enough for many language constructs that exist in modern programming languages. These include: global and local variables, exceptions, abnormal termination, breaks out of loops, procedures, as well as expressions with side-effects. Simpl also has theories for reasoning about the heap and references, thus allowing for the expression of linked data structures.

Essential elements of a typical Simpl theory include states, procedure declarations, and Hoare triples. The state takes the form of a `hoarestate` statement, which contains the list of variables used in the Hoare triple—examples will be given further below. Procedures are declared using Simpl's `procedures` declaration and have the following form:

```
procedures
  N (x::τ¹, y::τ², …| z::τ³)
  where v::τ⁴ … in B
```

where `N` is the procedure's name, `x` and `y` the formal parameters, τ^n a type, `z` the return value, `v` a local variable, and `B` the body. A `procedures` declaration is syntactic sugar for a number of deductive elements that are dynamically generated by Simpl and include a `locale`[1] and a `hoarestate`. All such locales are named using the name of the procedure and the prefix `_impl`.

Hoare triples have the usual form and in Simpl are written as:

```
Γ, Θ ⊢  {|P|} B {|Q|}, {|R|}
Γ, Θ ⊢t {|P|} B {|Q|}, {|R|}
```

for partial and total correctness, respectively. `Γ` is the procedure environment, `Θ` is a set of Hoare rules used as assumptions, `P` is the precondition, `B` is the body, and `Q` and `R` are the postconditions for normal and abrupt termination, respectively.

2.3 JML4

JML4 [5] is a next generation research platform for JML. It is an Eclipse-based Integrated Development and Verification Environment (IVE)—see Figure 2. Users can write their Java programs, annotate them with JML specifications, and prove them correct within the same environment using RAC, ESC, or FSPV.

Currently, JML4 supports JML's non-null type system (both statically and at runtime), the ability to read and make use of the extensive JML API library specifications, and basic RAC. Our research group, in addition to contributing to the basic infrastructure of JML4, is focusing on a new static verification component called the JML Static Verifier (SV). The JML SV offers support for ESC and FSPV. We examine the FSPV component in more detail in the sections that follow.

3. JML4 FSPV THEORY GENERATOR

In this section we present FSPV TG. Central to FSPV TG is a translator that takes Java programs along with their associated JML specifications and generates one or more Simpl theory files.

The choice of Simpl as a target VC language for our FSPV tool is motivated by two main reasons. Firstly, the generation of the VC is fully captured within Simpl, which as mentioned above, has been proven sound and complete. The alternative (and the norm) is to programmatically define VC generation and in some cases prove soundness, most of the time this is done by hand. Secondly, Simpl's syntax is such that rather than expressing lemmas as "low-level" VCs, we express them directly as Hoare triples.

At its current level, the FSPV TG supports a handful of JML and Java language elements, including method calls. Type-wise, only Integers and Booleans are supported while initial support for class related elements such as fields and methods are in place. A functional set of Java statements and expressions are supported. These include local-variable declarations with initialization and conditional and while-loop statements. Most arithmetic, relational, and logical operators are supported, including those with side-effects. Lightweight JML contracts and loop annotations are supported. All these elements are translated into Isabelle/Simpl using FSPV TG's translator.

FSPV TG's current translation phases, along with their individual inputs and outputs, can be seen in Figure 3. The first phase is named *TheoryTranslation*. The input to the first phase is the *JML+Java Abstract Syntax Tree* (AST) for a compilation unit. A compilation unit contains AST nodes for type declarations, which in turn contain type member nodes such as method declarations, fields, etc. The result of this phase is a (generic) *Theory AST* (Figure 4). This resulting theory AST consists of both a list of variables containing field-related information and one or more lemma AST nodes. Each lemma node is a translation of a single Java method declaration and represents the proof obligation for that method. Proof of the lemma establishes the correctness of the method with respect to its specification. A lemma node is a pair of:

[1] A locale is Isabelle's construct for parameterized theories

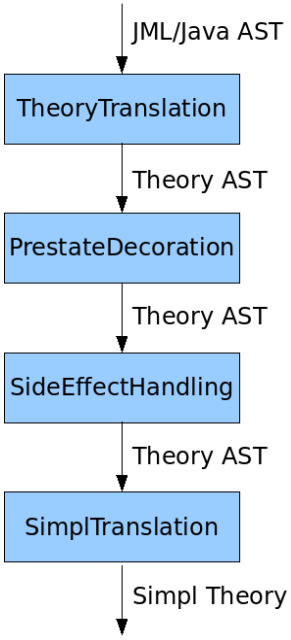


Figure 3: FSPV TG Phases

- a variable list containing all parameters and local variables declared in the method
- a Hoare triple containing the translations of the JML pre- and post-conditions, as well as the translation of the method body.

In the next phase, *PrestateDecoration*, the Theory AST is decorated with pre-state information. This entails storing the pre-state of the method parameters (since they can be modified within the method body) as well as the handling `\old` JML expressions. The result of this phase is an enriched Theory AST with additional variables, assignment nodes, and simplified `\old` JML expressions.

Additionally, in this stage we perform data analysis of the code in the presence of while loops. Translating while loops requires some care. Simpl adopts the classic Hoare rule for while loops whereas in JML, the assumption is that only a while body's assignment targets are "havocked"[2]—all other variables are assumed to remain unchanged. As such, the loop invariant is augmented to maintain additional state information—i.e., constraints that the non-havocked variables remain unchanged. Examples of this are presented in Section 4.

The third phase is called the *SideEffectHandling*. This phase translates expressions with side effects into a more palatable form, based on examples from the Simpl distribution of simplifying such expressions. To allow for this translation we introduce additional variables and assignment statements that hold intermediary results. To illustrate this, consider the following Java statement containing an expression with side-effects:

```
a *= b - i++;                                    (1)
```

It is translated into the following sequence of Java statements:

```
a0 = a;
i0 = i;
i = i + 1;
a = a0 * (b - i0);
```

This will translate into Isabelle using Simpl's notion of a *binder variable*: the expression *E′* `>>` *v* `.` *E*(*v*) evaluates to *E*(*v*) in which *v*, if it occurs free, will have the value *E′*, i.e. *E*(*E′*). The Simpl translation of (1) is:

```
a >> a0.
  i >> i0.
    i :== i + 1 ;;
    a :== a0 * (b - i0)
```

The last phase, called *SimplTranslation*, is responsible for generating the Simpl theory. For each theory AST, an Isabelle theory is created containing a `hoarestate` block for static and instance fields. For each lemma Theory AST node, a Simpl procedure and an Isabelle `lemma` block statement is created. The procedure contains the translation of the Java method into Simpl, while the lemma is there to prove the method correct with respect to its specification.
Examples will be given in Section 4.

Variables	$y : V$
Integers	$n : Z$
Boolean	$b : \{T,F\}$
Operators	op ::== + | - | * | / | ∨ | ∧ | = | != | ++ | -- | += | -= | *= | /= | :=
Expressions	e ::== $y \mid n \mid b \mid e\ op\ e \mid op\ e \mid e\ op$
Statements	s ::== y := e | **WHILE** e **INV** e **VAR** e s | **IF** e **THEN** s **ELSE** s | s ; s
Types	$\tau : \Gamma$
Lemma	l ::== $(y :: \tau)^*$ $\{e\}\ s\ \{e\}$
Theory	t ::== $(y :: \tau)^*$ l^*

Figure 4: Theory language abstract syntax

4. FSPV BY EXAMPLE

In this section, we present examples of recursive functions specified in JML and proven using Isabelle/Simpl. Each example allows us to highlight a particular capability of the JML4 FSPV TG or limitations in JML with respect to its linguistic ability to support the specification of recursive functions, especially for the purpose of proving total correctness. Note that for these examples, the resulting Simpl theories have a close resemblance to their associated Java classes. We found this quite pleasing since source code parts are easily identifiable in the corresponding theory.

4.1 Factorial

In Figure 1, presented earlier, we define the `Factorial` class with a single recursive method name `fac`, which returns the factorial of its integer argument. Our aim is to prove the method correct and that it terminates. The JML `measured_by` clause allows us to provide a measure that we can use to prove termination. A measure is a well-founded relation from a function to the natural numbers. Termination is achieved when the arguments of each recursive method call decrease with respect to the measure. While the definition of the `fac` method is simple, we note that it is already beyond the capabilities of ESC/Java2 due to the use of a generalized numeric quantifier in the method contract. Hence, factorial allows us to demonstrate the use, translation, and verification of JML's generalized numeric quantifiers such as `\product`. Moreover, ESC/Java2 does not support the `measured_by` JML statement. The corresponding

[2] Nothing can be assumed about the value of havocked variables.

```
theory Factorial imports Vcg SetHelper begin
  procedures
    Factorial_fac_int(n::int|result'::int)
  "IF ´n = 0
   THEN
     ´result' :== 1
   ELSE
     CALL Factorial_fac_int(´n - 1) ≫ n1.
     ´result' :== ´n * n1
   FI"
  lemma (in Factorial_fac_int_impl) Factorial_fac_int_spec:
  "∀n σ.
     Γ ⊢\<^sub>t {|σ. n = ´n ∧ ´n ≥ 0|}
     ´result' :== PROC Factorial_fac_int(´n)
     {| ´result'=(∏ {j :: int. ((1≤j) ∧ (j≤n))}) |}"
    apply(hoare_rule HoareTotal.ProcRec1
          [where r="measure (λ (s,p). nat \<^bsup>s\<^esup>n )"])
    apply(vcg)
    apply(auto)
  done
end
```

Figure 5: Simpl Theory for Factorial

Simpl theory is generated as part of the compilation process when the user selects the appropriate JML4 compiler options. The theory generated for Factorial is given in Figure 5.

The theory has two main parts: a Simpl `procedures` and an Isabelle `lemma` declaration. If more methods had been present in the Java class declaration then additional pairs of `procedures` and `lemma` declarations would have been given, one for each method. The `procedures` declaration contains the translation of the Java method in Simpl as well as all variables referenced by the program including `` `result' ``, a special variable added by the FSPV TG to hold the return value. The name of the class, the name of the method, and the method's signature are used to name the corresponding Simpl procedure. Encountering this procedure declaration, Simpl dynamically generates the `Factorial_fac_int_impl` locale that contains all the deductive machinery required for reasoning about the procedure. This locale is subsequently used in the `lemma` block to prove the procedure correct with respect to its specification.

We can identify the lemma definition enclosed within quotes. This definition follows the general format of a Simpl lemma definition proving total correctness (see Section 2.2)[3]. The lemma definition contains the Hoare triple to be proven, followed by its proof. We can clearly identify the pre- and post-condition at the top and bottom of the lemma enclosed within `{|` and `|}` character sequences which are used to denote assertions. Additionally, we bind the value of the input parameter to the logical variable `n` which is used in the postcondition in order to preserve the pre-state value of `´n`. The logical variable `σ` represents the pre-state; `σ` is always generated though it is not used in the examples presented here. In between, is a call to the `Factorial_fac_int` procedure. It is worth noting how JML `\product` quantified expressions are translated to Isabelle/HOL's product definition `∏` using an Isabelle set comprehension to specify the range. Isabelle/HOL's set theory is typed and extensive. It allows for set comprehensions and ranges which are ideal when translating JML numeric quantifiers.

To prove this procedure correct and that it terminates we need to provide a well-founded relation and to prove that subsequent recursive calls are decreasing with respect to its arguments—for our factorial example this means that subsequent recursive calls

[3] The \<^sub>t is how ProofGeneral subscript characters. Unfortunately not all of Proof General's X-symbols are supported in the Eclipse plug-in.

```
public class McCarthy {
  //@ requires n >= 0;
  //@ ensures \result == (100 < n ? n-10 : 91);
  //@ measured_by 101 - n;
  public static int f91(int n) {
    if(100 < n)
      return n - 10;
    else
      return f91(f91(n + 11));
  }
}
```

Figure 6: Recursive McCarthy's 91 Method

```
theory McCarthy imports Vcg begin
  procedures
    McCarthy_f91_int(n::int | result'::int)
  "IF 100 < ´n
   THEN
     ´result' :== ´n - 10
   ELSE
     CALL McCarthy_f91_int(´n + 11) ≫ n1.
       ´result' :== CALL McCarthy_f91_int(n1)
   FI"
  lemma (in McCarthy_f91_int_impl) McCarthy_f91_int_spec:
    "∀n σ. Γ ⊢\<^sub>t
       {|σ. n=´n ∧ ´n≥0|}
       ´result' :== PROC McCarthy_f91_int(´n)
       {| ´result'=(if 100 < n  then  n - 10 else 91 ) |}"
   apply(hoare_rule HoareTotal.ProcRec1
    [where r="measure (λ (s,p). nat (101 - \<^bsup>s\<^esup>n) )"])
   apply(vcg)
   apply(auto)
  done
end
```

Figure 7: Simpl Theory for McCarthy's 91 Function

are made using smaller non-negative integer values. Isabelle/HOL provides us with such a mechanism via the `measure` clause. The measure clause for this particular example is just the input parameter and it has the following form: `measure λ(s,p). nat ˢn`. To introduce this measure to our proof we make use of the `HoareTotal.ProcRec1` rule and we instantiate the `?r` schematic [18] variable with the measure using the `where` theorem modifier.

To complete this proof we need to provide additional properties pertinent to the set comprehensions used in the post-condition. These are included as simplification rules in the `SetHelper` theory (imported by the `theory` statement) which is provided in Appendix A. Finally, we complete the proof using two applications of the `vcg` and `auto` methods.

To work with the theory we use Eclipse's ProofGeneral plug-in [1] which is a generic front-end for interactive theorem provers supporting Isabelle. It is through Proof General that we prove this theory correct following the proof steps described in the previous paragraphs.

4.2 McCarthy's 91 Function

Our next example contains an implementation of McCarthy's 91 function [15]. The `f91` method, seen in Figure 6, is defined over positive integers and returns 91 for all `n <= 100` otherwise it returns `n - 10`. The measure for the function is remarkably simple: `101 - n`. McCarthy's 91 function is interesting because of its use of nested recursion.

The FSPV generated theory is shown in Figure 7. Like in the previous example, a Simpl procedure and its associated Simpl specification lemma are generated. We prove correctness and termination within Eclipse using the associated Proof General plug-in. Despite the nested recursion we are able to verify the procedure correct and that it terminates with relative ease: i.e., by merely asking Simpl to generate the verification condition (vcg),

```
class Fibonacci {
  //@ public static native int fib_spec(int n);

  //@ requires n>=0;
  //@ ensures \result ==  fib_spec(n);
  //@ measured_by n;
  public static /*@ pure */ int fib(int n) {
    if(n == 0)
      return 0;
    else if (n == 1)
      return 1;
    else
      return fib(n-1) + fib(n-2);
  }
}
```

Figure 8: Fibonacci Method (using native `fib_spec()`)

which Isabelle's auto method is then able to discharge without further user intervention. Surprisingly, our proof in Simpl is simpler than the corresponding proof for a native Isabelle/HOL function definition of the 91 function presented in [13].

4.3 Fibonacci Numbers

Our next example is a recursive method that calculates Fibonacci numbers (see Figure 8). The difference with respect to the previous cases is that in this example we make use of the `native` JML feature, recently proposed by Julien Charles [6]. In essence, this feature declares pure JML methods without an explicit definition. The definition is instead provided using the underlying target logic that JML annotated Java code is translated to. This provides for a more natural way of proving recursive methods that have in their specification recursive method calls. Moreover, it allows us to illustrate the definition of Isabelle/HOL functions and their use within Simpl assertions.

Figure 9 presents the generated theory suitably edited to include a definition of `fib_spec()` and our modifications that prove the method correct and that it terminates with respect to its specification and its measure, respectively.

The Simpl procedure declaration of `Fibonacci_fib_int` contains the translation of the Java statements and expressions into Isabelle/Simpl. Notice how binder variables are used to store the intermediate results of the recursive calls.

The `fib_spec()` function is the definition of the corresponding native pure methods. We make use of the Isabelle special polymorphic value `arbitrary` which is used to denote an arbitrary value. This is required because Isabelle/HOL functions are total by definition—i.e. we underspecify the function for negative integers. For every Isabelle/HOL function two proof obligations are required to be satisfied: one for completeness and compatibility of patterns and another for termination [13]. Their respective proofs follow the definition. It is worth mentioning that Isabelle/HOL provides a simpler form of defining functions where both of these proofs are satisfied automatically, however, the default termination proof (based on lexicographic order) is not sufficient for the `fib_spec` function—hence, the use of the "long" form.

The final part of this theory is the specification lemma. The proof proceeds as in the previous cases where the `HoareTotal.ProcRec1` rule is used, instantiated by a well-founded relation (via `measure`) and followed by an application of the `vcg` and `auto` methods.

Supporting reasoning about pure model methods having contracts that fully capture their behavior is possible (see Figure 10). This can be accomplished by using inductive sets to encode

```
theory Fibonacci imports Vcg begin
  procedures
    Fibonacci_fib_int(n::int | result'::int)
  "IF ´n = 0
   THEN
     ´result' :== 0
   ELSE
     IF ´n = 1
     THEN
       ´result' :== 1
     ELSE
       CALL Fibonacci_fib_int(´n - 1) ≫ n1 .
       CALL Fibonacci_fib_int(´n - 2) ≫ n2 .
       ´result' :== n1 + n2
     FI
   FI"

  function fib_spec :: "int ⇒ int" where
  "fib_spec n =
    (if n = 0 then 0 else
      (if n=1 then 1 else
        (if n < 0 then arbitrary
          else (fib_spec (n - 1)) + (fib_spec (n - 2)))))"
  by(pat_completeness, auto)
  termination by (relation "measure (λn. nat n)",auto)

  lemma (in Fibonacci_fib_int_impl) Fibonacci_fib_int_spec:
  "∀n σ. Γ ⊢\<^sub>t
       {|σ. ´n=n ∧ ´n≥0|}
       ´result' :== PROC Fibonacci_fib_int(´n)
       {|´result'=fib_spec(n)|}"
  apply(hoare_rule HoareTotal.ProcRec1
    [where r="measure (λ (s,p). nat \<^bsup>s\<^esup>n )"])
  by(vcg,auto)
end
```

Figure 9: Simpl Theory for Fibonacci

```
class Fibonacci {
  //@ requires n>=0;
  //@ ensures \result == (n==0)? 0 : (n==1) ? 1
  //@  : fib_spec(n-1)+fib_spec(n-2);
  //@ measured_by n;
  //@ public static pure model
  //@                       int fib_spec(int n);

  //@ requires n>=0;
  //@ ensures \result == fib_spec(n);
  //@ measured_by n;
  public static /*@ pure */ int fib(int n) {
    ...
  }
}
```

Figure 10: Fibonacci Method with fib_spec() as a model method

the method contract and then proving that the inductive definition is functional.

4.4 Ackermann's Function

In the previous examples we have dealt with functions having trivial measures. In this section we illustrate a total termination proof for a recursive implementation of the Ackermann function [15] (see Figure 11) which has a non-trivial measure. This measure is a well-founded relation on pairs of non-negative integers. In the process we also recognize the inadequacy of the `measured_by` clause in specifying this measure. Once more we make use of a native pure JML method to specify the post-condition. As we shall see, its definition in Isabelle also helps in making the case of preferring natural numbers instead of integers when working with non-negative values.

The complete theory that includes our modifications is presented in Figure 12. In addition to the `procedures` and `lemma` declarations we have defined two Isabelle/HOL functions

```
public class Ackermann {
  //@ public static native int ack_spec(int n);

  //@ requires n >= 0 && m >= 0 ;
  //@ ensures \result == ack_spec(n,m);
  public static int ack(int n, int m) {
    if(n == 0)
      return m + 1;
    else
      if(m == 0)
        return ack(n-1, m);
      else
        return ack(n-1, ack(n, m-1));
  }
}
```

Figure 11: Ackermann Method

(`ack'` and `ack_spec`) and a `lemma` declaration (`distrib_minus_int`) that proves that Isabelle's `nat` operator distributes over subtraction of integers, where the right hand side of the subtraction is the integer 1.

The `ack_spec` function is implemented over integer values that return the Isabelle `arbitrary` value when either one of its arguments is a non-negative number—in all other cases it makes use of the value returned by the `ack'` function. The `ack'` function is an implementation of the Ackermann function over natural numbers. It is possible to avoid writing the `ack'` function altogether and incorporated the remaining cases in the `ack_spec` definition—in fact our first attempts in a definition of the native method followed this approach. We were successful in completing an integer only definition of `ack_spec`. However, when this is used within the `Ackerman_ack_int_int_spec` lemma the Isabelle simplifier enters what it seems an infinite loop. In general, natural number based definitions are easier to work with in Isabelle/HOL. Hence, by using a natural number implementation of the Ackermann function as a first step we are able to prove the corresponding Simpl procedure correct. We are confident that even with our original approach a proof of correctness is achievable given additional investment on our part.

In the `Ackermann_ack_int_int_spec` lemma we have manually inserted the measure using the `HoareTotal.ProcRec1` rule as to demonstrate that Isabelle/Simpl is capable of proving termination of the Ackermann function. The measure we provide is in fact a list of two measures. As such they do not correspond to the current syntax and semantics of the `measured_by` clause. In Isabelle/Simpl such measure lists are specified using the `measures` combinator. This `measures` combinator is a generalization of the measure clause and it constructs a well-founded relation from a list of measures—it is explained in detail in [3]. We continue the proof with a set or repeated applications of the `auto` and `vcg` methods. These methods generate subgoals that each is resolved by cases on the `nat` type followed by an extra application of the `auto` method.

Table 1: A Comparison on Java's FSPV Tools

	LOOP	JACK	Krakatoa Why	FSPV TG Simpl
Maintained	×	×	✓	✓
Open Source	×	✓	✓	✓
Proven Sound	✓	×	✓	✓[1]
Proven Complete	×	×	×	✓[1]
Above two proofs done	in PVS	N/A	by hand	in Isabelle
VC generation done in prover	×	×	×	✓
Termination of recursive functions	×	×	×[2]	✓

[1] Simpl is proven sound and complete. The translation to Simpl is not.
[2] See main text for a qualification of this mark.

```
theory Ackermann imports Vcg begin
  procedures
    Ackermann_ack_int_int(n::int, m::int|result'::int)
  "IF ´n = 0 THEN
     ´result' :== ´m + 1
   ELSE IF ´m = 0 THEN
     ´result' :== CALL Ackermann_ack_int_int(´n - 1, 1)
   ELSE
     CALL Ackermann_ack_int_int(´n, ´m - 1) ≫ m1.
     ´result' :== CALL Ackermann_ack_int_int(´n - 1, m1)
   FI FI"
  function ack' :: "nat ⇒ nat ⇒ nat" where
    "ack' 0 m = Suc m"
  | "ack' (Suc n) 0 = ack' n 1"
  | "ack' (Suc n) (Suc m) = ack' n (ack' (Suc n) m)"
  by (pat_completeness,auto)
  termination
    by(relation "measures [λ(n,m). n, λ(n,m). m]",auto)
  fun ack_spec :: "int ⇒ int ⇒ int" where
    "ack_spec n m =
      (if n<0 then arbitrary else
       (if m<0 then arbitrary else
         int (ack' (nat n) (nat m))))"
  lemma nat_distrib_minus_int [simp]: "∀x. nat (x - 1) = (nat x) - (nat 1)"
   by (auto)
  lemma (in Ackermann_ack_int_int_impl) Ackermann_ack_int_int_spec:
    "∀n m σ. Γ ⊢\<^sub>t
       {|σ. n=´n ∧ ´n≥0 ∧ m=´m ∧ ´m≥0 |}
       ´result' :== PROC Ackermann_ack_int_int(´n, ´m)
       {|´result' =  (ack_spec n m) |}"
   apply(hoare_rule HoareTotal.ProcRec1
      [where r="measures [λ(s,p). nat \<^bsup>s\<^esup>n,
                          \<lambda>(s,p). nat \<^bsup>s\<^esup>m]"] )
   apply((auto|vcg)+,case_tac "nat n",auto,case_tac "nat n",auto)
   by (case_tac "nat m",auto)
  end
```

Figure 12: Ackermann Theory

5. RELATED WORK

In this section we examine three existing FSPV tools.

LOOP. The LOOP tool [12,21] was developed at the University of Nijmegen in Netherlands. LOOP covers a functional subset of sequential Java. In particular, LOOP can handle all of Java Card. Thus, LOOP is able to reason about expressions with side effects, exceptions, inheritance, and overloading. To our knowledge only multi-threading, inner classes and termination of recursive programs are left out.

The LOOP tool is a compiler. Its input is JML-annotated Java source code and its output is theories for the PVS theorem prover. These theories, along with a set of theories named "the prelude," are used as input to the PVS theorem prover when a developer wishes to conduct a verification session. The prelude contains the semantics of both JML and Java. Through user interaction, properties of these JML/Java sources can then be verified. A user working with LOOP-generated theories has a choice between a Hoare logic and two weakest-precondition calculi.

As compared to Isabelle/Simpl, LOOP's Hoare logic has been proven sound using PVS, but not proven complete. To our knowledge, the LOOP tool does not support termination of recursive programs. LOOP incorporates the semantics of JML and Java in its compiler generating primitive formulas which are then used as input to the PVS prover. FSPV-TG, on the other hand, generates Simpl theories which incorporate the semantics of sequential programming languages in terms of Hoare logic and weakest precondition semantics—i.e. the transformation from a Hoare triplet to a primitive formula is done within the prover.

JACK. The Java Applet Correctness Kit (JACK) tool [2] is an Eclipse plug-in. Like LOOP, JACK also translates Java programs into one or more theory files. However, JACK generates theories in a Java-like language called *Java Proof Obligation* (JPO) language. These obligations are generated using weakest precondition semantics which, to our knowledge, has yet to be proven sound. JACK provides support for a number of theorem provers, namely Coq, PVS, B, and Simplify—with Coq and Simplify being the most fully supported. Prover-specific theories are translated using the JPO theories as input. Additionally, JACK supports specification and verification at the bytecode level. Bytecode verification also makes use of a weakest-precondition semantics. In this case, this semantics is proven sound using the pen and paper approach [19].

The differences between the underlying logics of JACK and FSPV TG are similar to those of LOOP. JACK generates primitive formulas in Java, while we make use of Simpl's Hoare rules and weakest precondition semantics to generate the primitive formulas. Additionally, JACK does not support termination proofs for recursive functions.

Krakatoa. Krakatoa is an FSPV tool for JML annotated Java classes. Originally designed to generate theories for the Coq theorem proven it has recently been modified to output programs for the Why tool as well [11].

Why is a multi-tool Verification Condition (VC) generator. The input 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 [10]. The output is one or more theories for a number of provers. These include the automated Yices, CVC3, and the Interactive Coq, Isabelle, and PVS. It is worth noting that Why is general enough that it is used by Caduceus—a front-end for verifying C programs.

Krakatoa is similar to FSPV TG in the sense that it translates Java programs into an intermediate program. However, Why programs are translated into a prover-specific theory using the Why compiler written in Objective CAML. Consequently, it suffers from the same issues as LOOP and JACK with respect to having VCs generated programmatically. Krakatoa does not support reasoning about the termination of recursive methods as indicated by [16]. Nonetheless its underlying intermediate language, Why, does have support for specifying recursive functions (via the `rec` keyword) with measures (via the `variant` keyword).

Table 1 presents a comparison in terms of the soundness and completeness of the underlying logical foundations of these FSPV tools along with our own FSPV TG. Additionally, we report (second to last row) on which tools programmatically generate VCs and which generate them through a theorem prover. Finally, in the last row, we report on tool support for proving termination of recursive programs.

6. CONCLUSION AND FUTURE WORK

We have presented initial work we have done in implementing an FSPV tool in JML4. This FSPV tool makes use of Simpl—a logic for expressing and verifying sequential imperative programs developed within Isabelle/HOL. Simpl's Hoare logic has been proven sound and complete with respect to the programming language semantics. We have illustrated the current level of support that the FSPV TG provides and presented a sample of our experimental test cases. We have focused our attention on proving recursive programs correct and that they terminate.

We have shown programs implementing Factorial and McCarthy's 91 function and how the FSPV TG, at its current state, can correctly prove total correctness. We examined more complicated cases such as Fibonacci and the Ackermann function. In there we employed the recently introduced native feature that allows separating declaration and definition of JML pure methods. This separation allowed for an "easier", a more natural, and a flexible definition of the pure method in the underlying logic. Moreover, we have exposed inadequacies of JML in specifying complex measures such as the one for the Ackermann function.

Through our experiments we believe that we have demonstrated the feasibility of Isabelle/Simpl as a backend proving apparatus for our FSPV TG tool proving recursive programs correct and that they terminate. To our knowledge FSPV TG is unique with respect to applying Hoare logic rules and weakest precondition semantics within an interactive theorem prover.

We reviewed a number of related FSPV tools and we have seen that Simpl is the only logic proven both sound and complete within an interactive theorem prover. Additionally none of our reviewed tools supports total correctness of recursive programs.

We have plans for a number of future additions to this tool. A short-term goal is to make progress towards using pure model methods rather than native methods to specify recursive functions like the one given in our Fibonacci example. We will also be exploring extensions to the `measured_by` syntax of JML so that measures for Ackermann's function can be defined within JML directly.

REFERENCES

[1] Aspinall, D. et al. 2006. Proof general in Eclipse: system and architecture overview. ACM, 45-49.

[2] Barthe, G. et al. 2007. JACK - A Tool for Validation of Security and Behaviour of Java Applications. *In 5th International Symposium on Formal Methods for Components and Objects (FMCO),* , 152-174.

[3] Bulwahn, L. et al. 2007. Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL. In *Theorem Proving in Higher Order Logics*. 38-53.

[4] Burdy, L. et al. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf. 7*, 3, 212-232.

[5] Chalin, P. et al. 2008. JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML. In *Verified Software: Theories, Tools, Experiments*. 70-83.

[6] Charles, J. Adding native specifications to JML. *Formal Techniques for Java-like Programs,* , 2006.

[7] Cheon, Y. and Leavens, G.T. 2002. A runtime assertion checker for the Java Modeling Language (JML). In *International Conference on Software Engineering Research and Practice (SERP '02)*. CSREA Press, Las Vegas, Nevada, 322--328.

[8] Cok, D.R. and Kiniry, J.R. 2004. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. *In Construction and Analysis of Safe, Secure*

and Interoperable Smart Devices: International Workshop, CASSIS 2004 3362, , 108--128.

[9] Dijkstra, E.W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8, 453-457.

[10] Filliâtre, J. 2003. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program. 13*, 4, 709-745.

[11] Filliâtre, J. and Marché, C. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification*. 173-177.

[12] Jacobs, B. and Poll, E. 2004. Java Program Verification at Nijmegen: Developments and Perspective. In *Software Security - Theories and Systems*. 134-153.

[13] Krauss, A. 2008. Defining Recursive Functions in Isabelle/HOL. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/functions.pdf.

[14] Leavens, G.T. 2008. The Java Modeling Language (JML). http://www.eecs.ucf.edu/~leavens/JML/.

[15] Manna, Z. 1974. *Mathematical Theory of Computation*. Mcgraw-Hill College.

[16] March, C. et al. The Krakatoa Tool for Certication of Java/JavaCard Programs annotated in JML. .

[17] Nipkow, T. 2008. Project Bali. http://isabelle.in.tum.de/bali/.

[18] Nipkow, T. et al. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer.

[19] Pavlova, M. 2007. *Java bytecode verification and its applications*. Ecole Superieure en Sciences Informatiques de Sophia Antipolis.

[20] Schirmer, N. 2005. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *Logic for Programming, Artificial Intelligence, and Reasoning*. 398-414.

[21] Van Den Berg, J. and Jacobs, B. 2001. The LOOP compiler for Java and JML. *Tools and Algorithms for the Construction and Analysis of Systems, number 2031 in Lect. Notes Comp. Sci*, , 299--312.

[22] Winskel, G. 1993. *The formal semantics of programming languages: an introduction*. MIT Press.

APPENDIX A

```
theory SetHelper imports Main begin
 lemma preToInv [simp]: "{x. 1 ≤ x ∧ x ≤ (0::int)} = {}"
 proof -
   show ?thesis by auto
 qed

 lemma setinterval_iff: "{x. a ≤ x ∧ x ≤ b} = {a .. b}"
  by auto

 lemma prodEqProdTimes:
  assumes n: "m - 1 < (n::int)"
  shows "∏ {x. m ≤ x & x ≤ n} = ∏ {x. m ≤ x ∧ x ≤ n - 1} * n"
 proof-
  let ?A = "{m .. n}"
  let ?B = "{m .. n - 1}"
  let ?C = "{n}"
  have abc: "finite ?B" "finite ?C" "?B Int ?C = {}" "?B Un ?C = ?A"
  using n by auto
  from setprod_Un_disjoint[OF abc(1-3), of "%x. x"]
  show ?thesis unfolding abc(4) setinterval_iff by simp
 qed

 lemma upperBoundRangeSetEq:
    "{x. m ≤ x ∧ x < n} = {x::int. m ≤ x & x ≤ n - 1}"
  by(auto)

 lemma prodEqProdTimesLess [simp]:
  assumes n: "m - 1 < (n::int)"
  shows "∏ {x. m ≤ x & x ≤ n} = ∏ {x. m ≤ x ∧ x < n} * n"
 proof -
  show ?thesis
  by (simp only: upperBoundRangeSetEq,
      rule prodEqProdTimes, rule n)
 qed
end
```

Adapting JML to generic types and Java 1.6

David R. Cok
Eastman Kodak Company Research Laboratory
1999 Lake Avenue
Rochester, NY 14650 USA
david.cok@kodak.com

ABSTRACT

Despite the current effort to implement the Java Modeling Language for Java 1.5, and in particular for generic types, there has been no analysis of the effect of such a transition on JML itself, nor of what language changes should be implemented to take best advantage of the features of current Java. This paper analyzes the interactions between JML and the new features of Java 1.5 and 1.6, and it proposes appropriate changes to JML. Many implementation details for JML tools can be handled by choosing an existing Java 1.5+ compiler as a base; however, there are adjustments to the typing of JML expressions that would be appropriate, and there are issues needing careful attention arising from refinements, autoboxing, lock ordering operations, specification of enhanced for loops, type erasure, and the runtime execution of specifications involving type parameters. The features are implemented experimentally in OpenJML, an OpenJDK-based implementation of JML.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

specification, JML, Java, generics

1. INTRODUCTION

The Java Modeling Language (JML) [7, 8] has been a successful, widely used specification language for Java programs. Many tools [2] have been generated and many research groups[1] have used it as a basis for research and experimentation. The JML2 tool suite was written and maintained for Java 1.4. Java 1.5, introduced in 2004, brought significant changes to Java, but the work to evolve JML tools to work with Java 1.5 stalled for lack of resources. That omission is only now being addressed by the building of new versions of JML on top of Eclipse [4] and OpenJDK [12], although those projects are not yet ready with released tools.

Not explicitly addressed in those tool-building activities, however, are the changes to JML itself that are needed to keep it aligned with the generic type and other capabilities of Java 1.5 and 1.6. This paper addresses that deficiency.

Versions of Java beginning with Java 1.5 introduced many new features into the language; we will consider the following here:

- generic types and methods
- enhanced for statement
- autoboxing and unboxing
- annotations
- varargs
- static import
- enum types
- `java.lang.SuppressWarnings`
- the Java compiler, AST and annotation processing APIs

This paper presents an assessment of JML with respect to these features of Java, recommending alterations and extensions to JML where that would be beneficial for the language. We also note areas where similar activities are underway in other groups and highlight aspects that would benefit from cooperation. The issues discussed are relevant to interface specification languages for other programming languages with generic types, but we consider them only in the concrete context of Java and JML.

2. ENHANCEMENTS TO JML

In the following subsections we discuss the impact on and propose alterations to JML to accommodate the evolution of Java. A JML tool that is built upon a Java compiler will be able to accommodate most language changes without difficulty. The implementation effort is reduced further if the Java compiler infrastructure is used for parsing and type-checking JML expressions as well. Nevertheless, there are several issues that must be attended to.

2.1 Generic methods and types

The most significant language addition in Java 1.5 was generic types and methods. Type names in a JML specification may now be parameterized with concrete types or type variables, and model methods may now have type parameters. This affects the JML grammar in ways corresponding to the changes in the Java grammar: alterations are needed in the definitions of *reference-type*, *class-definition*, *interface-definition*, *class-extends-clause*, *name-weakly-list*, *method-decl*, and *primary-expr* (for method calls).

For tools built on existing compilers (and maintained by others) the infrastructure needed to support generic types in JML comes

[1]Publications from a variety of groups using JML are given at `http://www.jmlspecs.org`.

with the Java compiler. Parameterized model methods and parameterized model types by themselves pose no new difficulties.

2.2 Type parameters

However, the body of a class or method now has some additional type names in scope—those of the type parameters. In the case of methods, the scope of these type parameters extends over the method specifications. Thus the name lookup procedure for method and type specifications must include the type parameters. Model methods also may now be generic, so signature matching and type resolution for model methods must be enhanced in the same way as it is for ordinary Java methods.

Type names are not common in method and type specifications since those are mostly expressions. However, they can occur in type literals, **`instanceof`** and cast expressions, in declarations in **`forall`** and **`old`** clauses in method specifications, and in quantified and set comprehension expressions.

Within type and method specifications, the properties of a particular type parameter must be ascertained. Java properties such as the methods defined for the type are determined just as in the program code. JML also needs to determine which specifications apply to a method of a type parameter. Those are defined as the union of the specifications from the types making up the upper bounds of the type parameter.

2.3 Refinement

The first significant departure from what a Java compiler provides is in resolving refinement. JML allows the specifications for a class to be in a separate file from the source itself. In fact, the source may not even be present, since we may be specifying a binary file. In addition, there may be more than one refinement file for a given class. Refinement resolution consists of attaching each specification to the correct Java construct. So, declarations in specification files must be matched to Java declarations or entities in binary files.

The first task is to match class declarations. This is straightforward since there is at most one class with a given fully qualified name. There remains to be sure that the type parameters of the class in the specification file match those defined in Java, both in number and in any bounds restrictions. In general the parameter names may not be the same among the various class declarations, so a mapping of names may be required. It would be prudent to require that type parameter names be consistent among Java source and any JML specification files, simply to aid comprehension.

The second task is to match the declarations within a class. Fields have unique names, so they can be matched by name and their types checked for equality, taking into account type parameters. Method names are not unique and must be matched by signature, taking into account type parameters of the class and any type parameters declared for the method itself. (Thus refinement resolution must occur after the type name portion of a compiler's symbol table is built; it cannot be a simple textual match.) This sort of generic signature mapping is not needed in Java and must be implemented by the JML tool itself. It is made simpler, and does not restrict JML expressibility, if corresponding type variables have the same names.

Proposal: *The names of type parameters of a parameterized class, interface or method must be the same in all JML specification files for that program construct and must match those used in the Java source file, if that is available. (This not strictly necessary, but is a convenience for implementation.)*

2.4 Type specifications

The type variables of a generic class are in scope in type specification clauses such as invariants and constraints. It is conceivable that, like generic methods, one may want to parameterize axioms, invariants, constraints or initially clauses as well. The syntax would be straightforward and would take this form:

`axiom <T> (` *predicate* **`);`**

Of the possibilities, parameterized axioms would appear to be the most useful. Here is an example:

```
axiom <T> ( JMLObjectSet.<T>EMPTY().size() == 0 );
```

Experimentation may uncover good use cases that cannot be naturally expressed without parameterization. The implications for the encoding of specifications into the logic of target theorem provers are unknown, however, so it is best to leave this potential feature as experimental.

Proposal: *Reserve the following syntax for parameterized axioms, on an experimental basis, pending good use cases and practical experience:*

axiom-clause := `axiom` (*predicate* |
< *TypeParameter* [, *TypeParameter*]... > (*predicate*)) ;

where TypeParameter is a nonterminal defined in the Java Language Specification.

2.5 Method specifications

Method specifications are in the scope of any class or method type parameters, so type resolution needs to be applied just as it would be for the formal parameters or in body of the method. Although there are declarations in **`forall`** and **`old`** clauses, no additional parameterization of the specification appears to be useful.

There is one syntactic location where the JML grammar needs embellishment: the **`callable`** clause. This clause tells what methods may be called by the method at hand. Methods are denoted by their signatures, if necessary. Some disambiguation by type parameter may also be necessary. The current grammar for a **`callable`** clause contains a list of method names that has the following grammar, in part:

method-name-list := *method-name* [, *method-name*]...
method-name := *method-ref* [(*param-disambig-list*)]
| *method-ref-start* `. *`

We allow a *method-name* to be prefixed by an optional list of type arguments, as in

method-name-list := *method-name-gen* [, *method-name-gen*]...
method-name-gen :=
[< *ActualTypeArgument* [, *ActualTypeArgument*]... >] *method-name*
method-name := *method-ref* [(*param-disambig-list*)]
| *method-ref-start* `. *`

where *ActualTypeArgument* is defined in the grammar for Java.

Proposal: *Enhance JML to allow the syntax above for lists of methods in* **`callable`** *clauses.*

2.6 Specification expressions

2.6.1 `\TYPE`

The **`\TYPE`** type is JML's analog of the **`java.lang.Class`** type. Originally **`\TYPE`** was distinct from **`java.lang.Class`** in order to represent primitive types as well. However, Java evolved to represent primitive types as **`Class`** objects, so currently JML defines **`\TYPE`** as fully equivalent to **`Class`**. With the introduction of generic types, **`\TYPE`** could now be defined to be equivalent to **`Class<?>`**.

However, the Java runtime representation of class information erases any type parameter information: both **`List<Integer>`** and **`List<String>`** are simply represented as **`List<?>`**, for example. Thus, in order to retain the full information available statically, it would be better to define **`\TYPE`** as a fully reified combination of

the raw type information in **Class<?>** and the type arguments of a specific instantiation of the raw type; the behavior of **\TYPE** can be defined with appropriate axioms. Retaining this information would allow static checkers to warn about type-unsafe usage in Java that results in runtime exceptions.

In order to have the runtime JML behavior match the static analysis behavior, we need to define an executable representation of this combined information. One possibility is to have **\TYPE** encapsulate the **com.sun.mirror.type** classes. A restriction with this API is that it is meant to model the entities (e.g., types) in a specific declared program, rather than providing a facility to model types in general. Consequently it appears easier to model Java types straightforwardly as a separately declared executable class with appropriate specifications for static reasoning and implemented using Java APIs where they exist.

An expanded type system in JML may require some additional explicit type operators, such as the ability to extract a type parameter from a type object or to construct a parameterized type. However, there is insufficient experience with the specification needs with such a type system or with the proof rules that would be needed to propose a design at this time. We leave that for future work.

In the following sections we use **Class<?>** to mean Java's current runtime representation of class information (with erasure) and **\TYPE** to mean a representation in JML that reifies all statically declarable types.

Proposal: *Represent* ***\TYPE*** *as an entity distinct from* ***Class<?>****, reifying Java's raw type information and the type parameter information.*

2.6.2 \type

The specification expression **\type(**t**)** is currently defined by JML to be equivalent to t**.class** for a type name t. Java does not allow applying **.class** to a type name with parameters, as in **List<Integer>.class**. The runtime type literals do not retain the type parameter information, although it is used for parsing and typechecking. The expression **\type(List<Integer>)**, however, can be allowed. Thus it is consistent with the discussion of the previous section to define the type of **\type(**t**)** as **\TYPE**, allowing it to hold all of the type parameter information in an expression such as **\type(List<Integer>)**.

Proposal: *The type of* **\type** *is* **\TYPE***. The value of* **\type(**t**)** *is equivalent to* t***.class*** *for any unparameterized type name* t*. We allow* **\type(**t**)** *for parameterized type names, even though the types so represented cannot be expressed as Java class literals without erasure occurring.*

2.6.3 \typeof

The **\typeof** predicate returns the dynamic type (a value of type **\TYPE**) of its argument. The argument may be of primitive type. Its analog in Java is **Object.getClass()**. The value of **\typeof(**x**)** is

- undefined if x is null,
- equal to x**.getClass()** if x has nongeneric reference type, and
- equal to t**.class** if x has primitive type t.

We maintain the definition of **\typeof** as returning a value of type **\TYPE**. Then **\typeof** applied to an argument of parameterized type can include the additional type information that **getClass** erases.

Proposal: *The result type of* ***\typeof*** *is* ***\TYPE****; the expression is undefined if the argument is null.*

2.6.4 *subtype operation (*<:*)*

JML defines a binary operation **<:** between two **\TYPE** values meaning "is a subtype of". With the equivalence of **\TYPE** and **Class**, JML defined **t1 <: t2** as **t2.isAssignableFrom(t1)**. With the introduction of generics, **isAssignableFrom** no longer correctly models subtype relationships as seen by the compiler. We can define **<:** to act on two **\TYPE** values, but the runtime implementation of that operation must be separately implemented. Arguments that represent primitive types can be treated uniformly; a primitive type is not a subtype of anything but itself.

Proposal: *The arguments of* ***<:*** *still have type* ***\TYPE*** *and can include the* ***\TYPE*** *representations of primitive types. The operation is undefined if either argument is null.*

2.6.5 \elemtype

The **\elemtype** function takes an argument of type **\TYPE** and returns a value of type **\TYPE**. If the argument is an array type, the result is the component type of that array. This is equivalent to the method **Class.getComponentType** (for erased types). Consequently it is convenient to also define **\elemtype** to return null if the argument is not an array type, but undefined if the argument is null. Note that the argument is expected to be an array type, not an object of an array type. That is, the common use is, inconveniently, **\elemtype(\typeof(**o**))** for an object o, and not **\elemtype(**o**)**.

Proposal: *The argument and return types of* ***\elemtype*** *are* ***\TYPE****;* ***\elemtype*** *is undefined if the argument is null; the value of the expression is null if* x***.isArray()*** *is false for an argument* x*.*

2.6.6 \nonnullelements

The **\nonnullelements** predicate returns true if its argument is both non-null and an array object all of whose elements are non-null. It has been undefined if the argument is null or not an array object. The semantics can be improved with better typing, such as with the signature **\nonnullelements(Object[] t)** and corresponding signatures for each primitive type. No specifically generic method typing is needed. Multidimensional arrays are handled because any array is an instance of **Object**. The test for undefinedness (because the argument is not an array) is now changed: it was a semantic check on the argument's dynamic type, but now is simply a type check on the argument's static type.

Proposal: *Change the signature of the* ***\nonnullelements*** *function to be a set of overloaded functions with argument types of* ***Object[]*** *and* t***[]*** *for each primitive type* t*.*

2.6.7 *set comprehension and* JMLObjectSet

JML has a construct that allows the definition of new sets as expressions. For example, we can write

```
new JMLObjectSet {Integer i; o.contains(i); i > 0},
```

where o is a **Collection<Integer>**. The value of this expression is a **JMLObjectSet** that contains exactly the positive elements of o.

In current JML, the type of the result of a set comprehension is **org.jmlspecs.models.JMLObjectSet**, which is a set of **Object**s. However, the result type is in the process of being changed to **org.jmlspecs.lang.JMLSetType**, an interface defined in the core language package **org.jmlspecs.lang**. Any type that implements **JMLSetType** may be named in the constructor portion of the set comprehension expression. However, the type of the elements of the set is known from the declaration inside the set comprehension expression. The result should be a parameterized collection; in the example, this would be

```
JMLObjectSet<Integer> s =
   new JMLObjectSet {Integer i; o.contains(i); i > 0)}.
```

So, if C is the generic (without type arguments) type named after the **new** token and T is the element type named in the declaration, then the type of the result is C<T>, which then must implement **JMLSetType**<T>.

Proposal: *The JML model interface* ***JMLSetType<E>*** *is parameterized by the type of its elements. The set comprehension expression of the form*

new C ***{*** T ***e; ...; ...}***

has type C<E>*, where* E *is* T *if* T *is a reference type and is* T*'s boxed equivalent if* T *is a primitive type, and where* C<E> *must implement* ***JMLSetType***<E>*. The model types* ***JMLObjectSet****,* ***JMLValueSet****, and* ***JMLEqualsSet*** *would implement* ***JMLSetType****.*

2.6.8 \lockset

The value of the **\lockset** keyword is the set of all objects whose associated monitor is owned by the thread in which the **\lockset** expression is evaluated. It currently has type **JMLObjectSet**, but should now be **JMLSetType<Object>**.

Proposal: *The type of* ***\lockset*** *is* ***JMLSetType<Object>****.*

2.6.9 \max

The **\max** function takes a **JMLObjectSet** as an argument and returns an **Object**. Typically the argument is **\lockset**. The value of the expression is the object in its argument that is the largest (measured by the lock ordering operation) of all the elements in the argument set that are locked by the current thread. Corresponding to previous changes, the signature of the **\max** function is best expressed as

<T> T \max(JMLSetType<T> o) .

Proposal: *The signature of* ***\max*** *is*

<T> T \max(JMLSetType<T> o)*.*

The expression is undefined if the argument is null; the result is null if the argument contains no objects locked by the current thread.

2.6.10 *Autoboxing and the lock ordering operations* < *and* <=

JML overrides the less-than (<) and less-than-or-equal (<=) binary operations to apply to two **Object**s, returning a result according to a user-defined ordering. This feature interacts with Java's auto-boxing. Specifically, the operations are now ambiguous when the arguments are a primitive numeric type and its autoboxed equivalent: for **int i** and **Integer j**, **(i < j)** could be either the numeric comparison between **i** and the unboxed **j** or it could be the lock-order operation between the boxed **i** and **j**. The operations are also ambiguous between two numeric reference types: for **Integer i** and **Integer j**, **(i < j)** could be either the numeric comparison between the unboxed **i** and the unboxed **j** (as it would be in Java) or it could be the lock-order operation between the **i** and **j** (as it would be currently in JML without auto-unboxing).

Proposal: *This issue is currently under discussion*[2]*, but the favored resolution is to deprecate* < *and* <= *as the lock-ordering operators, replacing them with the nonoverloaded new tokens* <# *and* <#=*.*

2.6.11 *autoboxing and class literals for* **\bigint** *and* **\real**

JML introduced two new types, **\bigint** and **\real**: **\bigint** is the set of infinite-precision integers; **\real** models the real numbers. Both are intended to provide infinite-precision quantities from mathematics to be used in specifications, rather than only the finite-precision types from programming languages. Although the relevant semantics has been a point of discussion, the definition of JML is simplest if both are interpreted as primitive types. Then we also need to define the boxed equivalents: **java.lang.BigInteger** and a new model type **org.jmlspecs.lang.JMLReal**, respectively.

JMLReal would have a specification that is appropriate for real numbers. Its runtime implementation necessarily needs to approximate the behavior of reals. Also, since there is no infinite-precision primitive integer, the difference between primitive and reference types for **\bigint** must be handled by the type checker, with the executable implementation using **BigInteger** for both.

For each primitive type there is a corresponding class literal. It is different but of the same type as the literal for its boxed type. Thus **int.class** and **Integer.class** are unequal but both have type **Class<Integer>**. The corresponding **Class** values are needed for **\bigint** and **\real**. **Class** objects are typically obtained using native methods from the underlying virtual machine, so one cannot create **Class** objects for new kinds of primitive types. However, we can model these new primitive types as **\TYPE** values.

Proposal: *Define* ***\bigint*** *and* ***\real*** *as primitive types. Define* ***java.lang.BigInteger*** *and* ***org.jmlspecs.lang.JMLReal*** *as the corresponding boxed object types, with auto boxing and unboxing conversions corresponding to the other primitive types. Model the literals for these primitive types as* ***\TYPE*** *values. Chalin et al. [3] has explored the implications of various semantics of numeric operations in more detail.*

2.6.12 \only_called

The arguments of the **\only_called** predicate are method signatures. These must now be allowed to be parameterized method signatures, with either specific types or wildcard types. The same syntax is used for the method signatures as in the **callable** method specification clause.

Proposal: *The arguments of* ***\only_called*** *are now instances of* method-name-gen *as defined in section 2.5.*

2.7 Annotations

The annotation feature was a second major change in Java 1.5. In this case existing usage was not changed, but a new capability was created for describing properties of program constructs, and many groups began experimenting with annotations expressing type constraints. In conjunction with annotations, the Java framework provides an API to process annotations as part of compilation. With this API, additional syntactic or semantic checks can be performed that are not part of the compiler (or of pure Java). A number of annotation-related projects may influence the future of JML:

- JML tools are already experimenting with replacing modifiers in declarations (e.g., pure, non_null) with equivalent annotations (**@Pure**, **@NonNull**) from a JML-specific annotation package: **org.jmlspecs.annotations**.

- Taylor [1] experimented with using annotations for all JML specifications. This requires specification expressions to be String arguments to annotations. The approach is feasible but incurs different usability issues than the current JML design. Just as current JML must process comments containing extensions to Java expressions, an annotation-based specification language would need to parse and type-check the String arguments of annotations as extensions to Java expressions. There is currently no compiler or annotation processing support for this language processing.

[2] on the mailing list jmlspecs-interest@lists.sourceforge.net

- The JSR-308 project [9, 11] seeks to allow annotations in conjunction with any use of a type name in Java. This would allow annotations to be used as type modifiers. Then subtypes such as non-null types or readonly types could be easily defined and used uniformly; checkers for them could be built using the annotation processing API, as pure extensions to Java (as has been demonstrated).

- The JSR-305 project [10] seeks to standardize the naming of annotations. Currently, similarly named annotations are used by different groups for similar purposes, but with some differences in semantics. For example, JSR-305 defines **`@NonNull`**, **`@CheckForNull`**, and **`@Nullable`** as three different nullity related annotations, where JSR-308 and JML use just two: **`@NonNull`** and **`@Nullable`**, and IntelliJ uses **`@NotNull`** and **`@Nullable`**. This project would enable the expression of many very specific custom specifications; some examples are that the return value of a method should not be ignored, that a numeric value is positive, that a numeric value is nonzero, and that a collection (or string or array) is not empty.

Proposal: *JML should migrate to using annotations instead of modifiers, particularly if JSR-308 is adopted. (If not, current JML syntax will need to be retained, at least for those syntactic locations where annotations are not allowed.) JML should continue to investigate using annotations for a broader range of specifications.*

In addition the JML community needs to engage with the broader static analysis community in the following ways:

- JSR-308 will allow annotations to be used in more places than they currently are and will allow annotations to replace JML modifiers. It should be supported by the JML community.

- Continue investigation into allowing annotations in other locations in order to support current JML specifications. JML currently allows specifications as statements within method bodies, statement modifiers, and declarations within classes.

- Common fully qualified names for annotations as advocated by JSR-305 would be a good thing, as long as the semantics are also the same. Using the same (unqualified) annotation names with different semantics for different tools is a nuisance, or even with the same semantics but in different packages. There is not yet consensus on the appropriate semantics for each standard annotation name (for example, for nullity annotations).

- JML provides a general mechanism to express a large family of specifications, with a goal of a broad view of static analysis extending as far as possible toward software verification. Annotations in general, particularly if names are standardized by JSR-305, provide a means to define many specific annotations, with a goal of enabling best-effort checks of commonly used, quite specific, specification predicates. It is an open question about how these two approaches should coexist and what combination provides the best and most usable tools for the software developer and specifier.

Proposal: *The JML community should engage more vigorously with both JSR-305 and JSR-308 to enable outcomes that are mutually beneficial and allow a good migration path for JML.*

2.8 Other general changes to Java

2.8.1 Static import

The Java static import statement allows a compilation unit to use static names from another class without qualifying them with a class name. JML has a model import statement corresponding to Java's import. The types imported by a model import statement are only available in JML statements and not in the Java program itself. With the introduction of Java's static import, JML's model import should also have a static option.

This is expected to have little effect on JML implementations. In fact most implementations to date do not distinguish Java from JML imports—all imported names are available in both parts of a compilation unit. This is an incompleteness in the JML implementations, but it rarely causes trouble and problems can be worked around by using fully qualified names.

A proper implementation of JML's model import needs to keep two namespaces of imported names: the Java namespace and the Java+JML namespace. Neither the OpenJDK nor the Eclipse Java compilers can be readily extended to do this. However, the problem is no more difficult with the addition of static imports.

2.8.2 Enum types

The Enum type facility adds true type-safe Enum types to Java. Presuming a JML implementation can use a Java compiler to parse and typecheck JML expressions, this feature is available to JML tools without additional implementation effort.

2.8.3 varargs

The varargs feature allows the declaration of methods that take an arbitrary number of arguments (of the same type). A Java compiler will also provide this capability to JML tools. Keep in mind that model methods should also have the varargs feature. Typically these are parsed in the same way that Java methods are.

2.8.4 Enhanced for statement

The enhanced for statement causes no difficulties for JML tools in itself. However, specifying it is a problem. Traditional while, do, and for statements have an iteration variable that is available to and almost always needed by the specifier in writing loop invariants. For example, a simple for loop might be specified as follows:

```
int sum = 0;
//@ loop_invariant 0<=i && i<=10;
//@ loop_invariant sum == i * (i+1)/2;
for (int i=0; i<10; i++) {
    sum = sum + i;
}
```

It is important to have the loop variable **`i`** available, so that the invariant can be written in terms of the iterations already completed. The loop variant also needs the loop variable to be able to show that the loop is making progress toward termination.

The enhanced for loop provides no such variable. An example of such a loop is this:

```
int[] array = ...
int sum = 0;
for (int element : array) {
    sum = sum + element;
}
```

The loop invariant we would like to write is

```
sum == (\sum int k; 0<=k && k<i; array[k]),
```

where **`i`** is the index of the next array element to be processed. But this value is not available. Java provides two types of enhanced for

statements: one takes an array, as in the example above, the other takes an object of type **Iterable**.

Spec# [6] has solved this problem by introducing a **\values** keyword whose value is a sequence of all of the values that have been iterated over so far. For Java and JML, I propose a corresponding solution, but with two keywords.

• Associated with each enhanced for loop is a new keyword **\index** of type **int**. The keyword represents the 0-based number of the current iteration. For enhanced for loops based on arrays this is also the index in the array of the current array value. The keyword is in scope within the body of the loop and in the loop specifications just prior to the loop. The value of **\index** begins at 0 and increments until equal to $array$**.length**, for array-based loops. The keyword may not be assigned to. Thus the example above is equivalent to (were **\index** a valid Java variable)

```
int[] array = ...
int sum = 0;
for (int \index = 0; \index < array.length; \index++) {
    int element = array[\index];
    sum = sum + element;
}
```

We can specify our example as follows

```
int[] array = ...
int sum = 0;
/*@ loop_invariant sum ==
          (\sum int k; 0<=k && k<\index; array[k]); */
//@ decreasing array.length - \index;
for (int element : array) {
    sum = sum + element;
}
```

• A second new keyword is **\values**. This keyword would have type **org.jmlspecs.lang.JMLList<**T**>**, where T is the type of the iteration variable and **Iterable<**T**>** is the type of the iteration collection. The value of **\values** is a sequence of the values returned so far (prior to the current iteration) by the iterator (autoboxed if the loop is an array-type loop and the array element type is a primitive type). Thus

```
Set<Integer> set = ... // all positive integers
int max = 0;
for (Integer i : set) {
    if (max<i) max = i;
}
```

would be specified as

```
Set<Integer> set = ... // all positive integers
int max = 0;
/*@ loop_invariant max == \values.size() == 0 ? 0 :
              (\max int k; \values.contains(k); k); */
for (Integer i : set) {
    if (max<i) max = i;
}
```

There are two alternatives for when the addition of the loop variable's value to the **\values** list occurs: (a) as part of the update step (after the loop body is executed), or (b) immediately after the value is extracted from the iterator. These two alternatives are being evaluated; the discussion below assumes the first design. Loop invariants are the same in both cases. However, in (a), **\index** always equals the size of **\values**, but in the body of an iteration, the current value of the loop variable is not yet in the **\values** list; if the loop is exited by a break statement, that value will not be in the list. In (b), extracted values are always in the list, but what is true in an invariant is not necessarily true in the body, since **\index** and **\values** are updated at different times. In the first design,

```
//@ loop_invariant ...
for (T element : array) {
    ... body ...
}
```

is equivalent to (where **T'** is **T** or its boxed equivalent)

```
int \index = 0;
JMLList<T'> \values = ... (empty list of T' )...
T element;
for (; \index < array.length ;
                      \index++, \values.add(element)) {
    ... check loop invariant
    element = array[\index];
    ... body ...
}
... check loop invariant ...
```

and

```
//@ loop_invariant ...
for (T element : iterable) {
    ... body ...
}
```

is equivalent to

```
int \index = 0;
JMLList<T> \values = ... (empty list of T )...
Iterator<T> iterator = iterable.iterator();
T element;
for (; iterator.hasNext() ;
                  \index++, \values.add(element)) {
    ... check loop invariant
    element = iterator.next();
    ... body ...
}
... check loop invariant ...
```

Note that if the **Iterable** collection is known to have fixed size, then something like **list.size() - \values.size()** makes an appropriate loop variant. However, the size of an **Iterable** is not necessarily known or fixed.

It is not strictly necessary to define both **\index** and **\values** since **(\index == \values.size())**. However, **\index** is more natural for loops that iterate over arrays, and it seems more readable and easier for reasoning engines to use, hence the proposal here is to define both keywords. If loops are nested, the **\index** and **\values** keywords in the inner loop will hide the corresponding keywords for an outer loop.

Proposal: *Define the keywords* ***\index*** *and* ***\values*** *for enhanced for loops with the semantics described above. Create a parameterized interface* ***JMLList<E>*** *in* ***org.jmlspecs.lang****.*

2.8.5 **SuppressWarnings** *and* **nowarn**

Java 1.5 introduced the **java.lang.SuppressWarnings** annotation as a mechanism for user control over compiler warnings. The arguments of the annotation name the warnings that then will not be issued in the context to which the annotation applies. JML has had

a lexical construct, **nowarn**, that offered similar capabilities. Thus the question: can **java.lang.SuppressWarnings** replace **nowarn**?

The short answer is partially. There is a key difference between the two constructs. The **nowarn** token is lexical; it may occur anywhere in the source code and applies to the source code line on which it appears. An annotation is constrained to appear in conjunction with declarations and packages; the **SuppressWarnings** annotation is allowed on type, field, method, constructor, parameter, and local variable declarations. It is also relevant that the **SuppressWarnings** annotation has only source retention and is unavailable at runtime. The JSR-308 proposal would expand the use of annotations to also be allowed on types anywhere they appear.

The warnings from JML tools are of two sorts. Some warnings are compiler-like: misuse of various language constructs. Where these are nonfatal, they can be suppressed just like compiler warnings might be. The more important warnings from JML are runtime or statically found assertion violations. These are associated with nearly every JML construct and implicitly with many Java language features. To be useful, a warning suppressor for JML needs (a) to be more fine-grained than at the method declaration level, and (b) to be able to be applied to any sort of specification construct. Thus the **SuppressWarnings** annotation is not currently an adequate replacement for **nowarn**, although it can provide similar functionality on a coarser scale. A Java annotation that could appear anywhere a comment could appear would be very useful for JML.

Proposal: *JML tools should recognize a common (to be agreed upon) set of warning names for various kinds of assertion violations and recognize their use in* ***SuppressWarnings*** *annotations. The* ***nowarn*** *construct should continue to be used (using the same set of warning names) and should not be deprecated for now.*

2.9 Model classes

JML contains a library of classes (in **org.jmlspecs.models**) intended to model mathematical constructs and to be useful in specifications. Consequently they are designed as types with immutable values and pure, functional methods. The classes have specifications suitable for static analysis and implementations that can be executed at runtime, although they are not necessarily efficient.

Many of these classes implement collections or related constructs and they should be rewritten as generic classes or interfaces, similar to the reimplementation of Java's collection classes, but retaining the design of immutable values. Specifically, the following should be reimplemented (here `#` stands for one of `Object`, `Equals`, and `Value`):

- Collection classes that should be parameterized by element type:
  JMLCollection, JML#Bag, JML#Sequence, JMLSetType, JML##Pair (e.g., JMLEqualsObjectPair), JML#Set, JML#To#Map, JML#To#Relation, JMLList#Node

- Other types needing generic parameters:
  StringOfObject, JMLComparable, JMLIterator, JMLModelObjectSet, JMLModelValueSet, JMLValueBagSpecs, JMLObjectSequenceSpecs, JMLValueSequenceSpecs, JMLValueSetSpecs

- Enumerations that should be converted to Iterators, with type parameters:
  JMLEnumeration, JMLEnumerationToIterator, JML#BagEnumerator, JML#SequenceEnumerator, JML#SetEnumerator, JML#To#RelationEnumerator, JML#To#RelationImageEnumerator

- Comparison operations needing type parameters (similar to **java.lang.Comparable**):
  org.jmlspecs.models.resolve.*CompareTo

Although not directly a result of the move to generic types, the model classes are not quite appropriately divided between the packages **org.jmlspecs.lang** and **org.jmlspecs.models**. The design is that classes in **org.jmlspecs.lang** are needed by the language features themselves. Consequently **JMLSetType** is there since it is the type of a set comprehension expression and **\lockset**, and **JMLDataGroup** is used for datagroups. In addition, **JMLIterator** and **JMLIterable** are used by **JMLSetType** and should be in **org.jmlspecs.lang**.

Proposal: *The model classes should be reimplemented with generic types. The generic versions should be placed in a new package,* ***org.jmlspecs.genericmodels****.* ***JMLIterator*** *and* ***JMLIterable*** *should be moved to* ***org.jmlspecs.lang****.*

2.10 Existing specifications

There are many JDK classes with at least partial JML specifications (although many more are needed). Many of those classes, particularly collection classes, became generic classes when the language moved from 1.4 to 1.5. The JML specifications for those classes now need to be ported as well. For the most part that work is straightforward, but there is one interesting aspect.

Most of the specifications for nongeneric collections include a ghost field **\TYPE elementType**, intended to hold the dynamic type of the elements of the collection. This is now superseded by the type parameter of the generic collection. Static analysis tools operating on source code can readily use the type information of type parameters. However, current Java erases the generic type information in binary classes; expressions such as **\type(E)** or **E.class** for a type parameter **E** are not legal. Thus runtime checking of JML will still need the **elementType** information.

A runtime assertion checker might correct this deficit by passing the type information into constructors and generic methods as additional parameters. A constructor expression such as, for example, **new HashSet<Integer>()**, would effectively be rewritten as **new HashSet$(Integer.class)**; the type parameter information would be stored in synthetic fields inside the class, equivalent to the **elementType** specification fields. A solution such as this is a matter for future research; it is expected to encounter tricky interactions among proof rules, generic type systems, and Java's current type erasure.

Proposal: *All of the existing JDK specifications need to be ported to Java 1.5. The* ***elementType*** *ghost field previously used in collection, enumeration and iteration types can be deprecated once generic type information is retained in compiled Java. Interim runtime assertion checking implementations can experiment with the auxiliary method parameter solution described above for accessing type parameter information at runtime.*

2.11 The compiler, syntax tree and annotation processing APIs

The compiler, compiler tree and annotation processing APIs together offer a promising step toward better future JML tool generation. The annotation processing API allows user code to process the parse trees of compilation units as parsing occurs. The tool can choose to process all files or only those that are marked with recognized annotations. New compilation units can be generated and entered into the parsing process. The compiler tree API allows the ASTs to be traversed and inspected, and the compiler invocation API allows programmatic control of the compilation process.

However, the current capabilities of these APIs are not yet sufficient for easy construction of JML tools.

- JML needs to parse Java-like expressions, obtaining ASTs representing expressions. Most of JML's expression syntax is the same as Java's, but there are some JML-specific extensions. There is as yet no facility either for extending the compiler or even for invoking the compiler on code fragments. The fact that the public API ignores Java comments, which is where JML specifications currently reside, is an additional complication.

- JML needs to be able to use and extend the name resolution and type checking capabilities of the Java compiler. Those compiler phases happen after annotation processing is performed, so there is currently no way (through the public API) to apply type checking to the JML specification expressions or to extend it for JML extensions.

- For runtime checking, a JML tool needs to modify the syntax tree to represent the source code with assertion checks included. The public APIs currently do not allow replacing one compilation unit with a revision nor revising a compilation unit's AST directly.

However, if the extended parsing and typechecking problems were resolved, the annotation processing API could enable static checking, so future developments in these APIs are worth following. Note that the functionality needed to implement static or runtime checking for JML can be created by direct extension of the publicly available OpenJDK source code, as tools such as the Checker framework [9] and the OpenJML [12] project have done.

3. IMPLEMENTATION AND FUTURE WORK

These enhancements to current JML are implemented on an experimental basis in the OpenJML project. OpenJML is a JML parser and typechecker built by extending the OpenJDK 1.6 source code. Porting the specifications of model types and the JDK is in progress.

Evaluation of JML's specification capabilities against industrial code is an ongoing activity that is being carried out in the context of some issues left unaddressed by this paper: the degree to which full reification of parameterized types is a needed design choice, the need for additional specification constructs for type manipulation, the relationship between the goal of full verification and checks of specific conditions (as, for example, by the FindBugs [5] tool), and the appropriate use of the evolving Java APIs for implementing static analysis tools.

4. CONCLUSIONS

The migration of JML to Java 1.5 and 1.6 has been mostly a task of accommodating the generic type facility of the recent versions of Java. The assessment described in this paper identified a number of areas where typing changes of JML features and a conversion to using generic types throughout JML would be beneficial. JML extensions are needed in the enhanced for statement and changes in the lock ordering operation; generics require some enhancements to refinement resolution; and some careful design work is needed to integrate JML's additional primitive types and to model reified and erased types statically and at runtime. Other changes to Java, such as annotations and some new public APIs, may provide benefits as they evolve, but are not ready to be used for implementing JML itself or to replace existing JML features. Finally, the usefulness of annotations and annotation processing has prompted a number of projects to adopt annotation processing for static analysis; the JML community should engage more fully with those efforts for mutual benefit.

5. REFERENCES

[1] K. P. Boysen. A specification language design for the Java Modeling Language (JML) using Java 5 annotations. Technical Report 08-03, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, Apr. 2008.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[3] P. Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, June 2004.

[4] P. Chalin, P. R. James, and G. Karabotsos. An integrated verification environment for JML: Architecture and early results. In *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 47–53. ACM, Sept. 2007.

[5] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, New York, NY, USA, 2007. ACM.

[6] B. Jacobs, E. Meijer, F. Piessens, and W. Schulte. Iterators revisited: Proof rules and implementation. In *7th Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2005.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[8] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, Mar. 2005.

[9] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 22–24, 2008.

[10] `http://jcp.org/en/jsr/detail?id=305`.

[11] `http://jcp.org/en/jsr/detail?id=308`.

[12] Unpublished information is available at `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/OpenJML/trunk/OpenJML/README`.

Using Analysis Patterns to Uncover Specification Errors

William Heaven Alessandra Russo
Department of Computing, Imperial College London
{william.heaven, ar3}@imperial.ac.uk

ABSTRACT

Developing or maintaining a formal software specification is a task unfortunately prone to the accidental introduction of logical errors, particularly inconsistencies. At worst, such errors can be dangerously misleading. For example, many software analysis tools that require a formal specification as input produce false positives when faced with inconsistency, making it more likely that developers miss errors in the software. At the same time, most existing analysis tools supporting specification development are not well suited to the detection of inconsistencies without explicit direction from an expert user. To address this shortcoming, this paper presents novel analysis "patterns" that can automatically guide specifiers through logical pitfalls of this kind by not only checking a given specification formula, but recursively checking the subformulae of that formula. By doing so, rather than present a specifier with potentially misleading feedback, use of these patterns can automatically ensure—without expert direction—that accidentally introduced inconsistencies are uncovered.

1. INTRODUCTION

Formal software specification largely remains the exclusive and sparsely-populated province of experts and enthusiasts due, in part, to the demands placed on the practitioner. Formal (typically declarative) specifications are prone to the accidental introduction of logical errors, particularly inconsistencies, during their development. Further, it is rarely the case that developing a specification is a task done once, checked, and forgotten. In practice, as software evolves, a specification must be extended with the addition of new constituent formulae, making the introduction of logical error an ongoing risk.

The difficulty is compounded by the fact that most specifications of non-trivial software systems typically contain many logical interdependencies and, therefore, the consistency of one part of a specification is likely to be affected by changes to other parts. Currently popular specification languages for component-based software (such as JML [15] and Spec♯ [1]) allow side-effect-free method calls to be used as terms in specification formula. So, for example, a side-effect-free method *size()* that returns an integer result could be used in a specification formula such as $size() < MAX$. However, while this specification-language feature has the great advantage of affording a succinct and modular specification style, it exponentially increases the number of dependencies between formulae in a specification. For example, the consistency of any formula containing the term *size()* will depend on the formulae in the method specification for *size()*, which may in turn be expressed using side-effect-free method calls and thus depend on the method specifications of those methods, and so on. Interdependencies of this sort, where the specification for a method like *size()* may not be immediately visible from the contexts in which the term *size()* is used in a formula, make it even harder to avoid introducing—and to notice—logical errors.

There are many tools available for specification analysis ranging from lightweight static- and runtime-checking tools to those that offer the potential for more heavyweight verification [1, 3, 16]. However, most concentrate on analysis of the relation between specification and code and not on analysis of the specification itself. Tools such as the SAT-based Alloy Analyzer [12] permit versatile analyses of specifications but even the Alloy Analyzer provides misleading feedback when analysing an inconsistent specification unless expertly directed. This is because the results of a consistency check will be positive (suggesting no inconsistency) when the set of formulae in question are not only consistent but vacuously consistent—"valid"—on account of inconsistent subformulae. For example, a formula $\phi \Rightarrow \psi$ is valid if ϕ is inconsistent. In this case, the logical error causing ϕ's inconsistency is "hidden" by the positive result of the consistency check.

What is needed are powerful automated tools to support the developers and maintainers of a specification. As a step towards this goal, this paper presents a set of analysis "patterns" that guide specifiers through the pitfalls of logical analysis by not only checking the consistency of a given specification formula but recursively checking the subformulae of that formula. By doing so, rather than present a specifier with potentially misleading feedback, use of these patterns can automatically ensure—without expert direction—that accidentally introduced inconsistencies are uncovered. An implementation of these patterns, using the Alloy Analyzer as a backend, has also been developed [10].

The analysis patterns are presented in the context of satisfiability-based analysis and, following a motivating example in Section 2, some preliminaries to their presentation are set forth in Sections 3 and 4. Section 5 then presents the patterns themselves. An implementation is briefly discussed in Section 6 before briefly considering some related work in Section 7. Finally, Section 8 concludes.

2. MOTIVATING EXAMPLE

One problem in evolving software is that it is often possible to adversely affect existing code by adding something new. Where that new thing is a subtype, ensuring that the subtype is a behavioural subtype [17] is a good way to avoid introducing undesirable behaviour. Behavioural subtyping effectively guarantees that the addition of a subtype does not affect the behaviour of the existing program. A behavioural subtype can be substituted for its supertype without observable difference in program behaviour. In specification languages such as JML and Spec♯, behavioural subtyping can be enforced via specification inheritance whereby the specification

of a subtype implicitly includes that of its supertype [5].

Consider a Java class *Queue* with a boolean method *insert()*. If *entries* is the data structure in *Queue* representing the queued elements, a postcondition for *insert()* might be specified in JML as follows (the JML specifications here are expressed using boolean Java expressions plus the standard propositional operators and the *old* keyword denoting pre-state values; they appear between special "/*@. . .@*/" comments):

```
/*@ ensures (result ==> contains(e))
        && (entries == \old(entries.add(e))); @*/
boolean insert(Entry e) { . . . }
```

This says two things. Firstly, that the boolean result of inserting element *e* implies the boolean result of a call to *contains()* on the same *Queue* object. Secondly, that *entries* after a call to *insert()* is equal to *entries* before the call in all respects other than *e*'s addition. In other words, the only difference between post- and pre-state *entries* is that *e* is added: all other elements in *entries* remain the same.

Assume that an evolution of the software containing *Queue* involves adding a subtype *BoundedQueue* which adds an extra method *size()* and overrides *insert()*. These additions might be specified as follows:

```
/*@ ensures size() == entries.size(); @*/
/*@ pure @*/ int size() { . . . }

/*@ also
    ensures size() < \old(entries.size()) && size() <= MAX; @*/
boolean insert(Entry e) { . . . }
```

The specification for *size()* says that its integer result will always equal the result of a call to the *size()* method of *entries*. Note also that *size()* is specified to be side-effect free with the keyword *pure*. This means that *size()* can be used as a term in the postcondition of the overriding *insert()*. The *also* keyword highlights that this new postcondition is considered in conjunction with the postcondition of the overridden *insert()*. Thus, the postcondition of *insert()* in *BoundedQueue* is the postcondition of the overridden method plus the above, which says that the size of the queue after a call to *insert()* is less than the pre-state value of *entries.size()* and less than or equal to some given value *MAX* (it is assumed, without going into detail, that when the queue is already full the insertion does not take place and the size of the queue does not change).

For *BoundedQueue* to be a behavioural subtype of *Queue*, the postcondition of *insert()* in *BoundedQueue* must imply the postcondition of *insert()* in *Queue*. This should be enforced by the fact that the overriding *insert()* includes the postcondition of the overridden *insert()* and the implication can be checked in a tool such as the Alloy Analyzer. As might be expected, the result given by the Alloy Analyzer in this case is that the implication is valid.

However, this positive result is misleading because it hides a logical error in the specification of *insert()* in *BoundedQueue*. This postcondition says that following a call to *insert()*, the size of the queue is *less than* the pre-state value of *entries.size()*, which is inconsistent with the specification of *size()* and the existing *insert()* postcondition. Following a call to *insert()* the size of the queue cannot be less than the pre-state value of *entries.size()*.

What is needed to uncover hidden specification errors in cases such as this is a means to analyse not only the top-level formula but also its subformulae. Specification errors are often not detected explicitly through a consistency check of top-level formulae alone. In this case, an automated analysis should detect not only that the implication is valid (vacuously consistent) but further investigate the subformulae of the analysed formula—the constituent formulae not only of the *insert()* method specification but also of the *size()* method specification—to uncover the source of the validity, which for an implication is possibly an inconsistent antecedent.

3. SATISFIABILITY VALUES

Establishing the consistency of a specification formula can be considered an instance of the Boolean Satisfiability Problem (SAT) [18, 8]. A formula ϕ in a specification language with a well-defined semantics is said to be *satisfiable* iff there is a possible assignment of values in that semantics to the terms of ϕ (variables, constants, and side-effect-free method calls) that makes ϕ true. Conversely, ϕ is said to be *unsatisfiable* iff there is no such assignment, i.e., for every possible assignment ϕ is false. Henceforth, **s** will denote the value *satisfiable* and **u** will denote the value *unsatisfiable*.

Satisfiability analysis in practice has well known limitations and automatic decision procedures for deciding satisfiability tend to be incomplete. While these limitations will be touched on briefly in Section 6, an ideal satisfiability procedure, or "oracle", will be assumed for clear and succinct presentation of the analysis patterns. This oracle is deemed to be sound and complete.

Definition 1 (Satisfiability Oracle). Let Φ denote the set of formulae in a specification language. An ideal satisfiability procedure, or *oracle*, is represented by the function $SAT : \Phi \rightarrow \{\mathbf{s}, \mathbf{u}\}$ such that, for a given formula $\phi \in \Phi$, $SAT\ (\phi) = \mathbf{s}$ iff ϕ is satisfiable and $SAT\ (\phi) = \mathbf{u}$ otherwise.

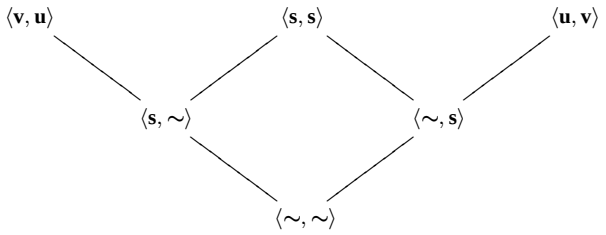
For a formula ϕ, $\mathbf{s}\ (\phi)$ will denote that $SAT\ (\phi) = \mathbf{s}$ and $\mathbf{u}\ (\phi)$ will denote that $SAT\ (\phi) = \mathbf{u}$.

The single query $SAT\ (\phi)$ is sufficient to decide whether $\mathbf{s}\ (\phi)$ or $\mathbf{u}\ (\phi)$. Further, if a formula ϕ is unsatisfiable, i.e., false for all assignments, then $\neg\phi$ must be true for all assignments. A formula that is true for all assignments is said to be *valid*. Therefore, to discover that ϕ is unsatisfiable is also to discover that $\neg\phi$ is valid. On the other hand, if a formula is only true for some assignments but not all, i.e., there are some assignments for which its negation is true, then the formula is said to be *contingent*. Thus, discovering that ϕ and $\neg\phi$ are both satisfiable is to discover that both are contingent. Finally, if a formula ϕ is known to be satisfiable, $\neg\phi$ cannot be valid because this would contradictorily require ϕ to be unsatisfiable. Therefore, a value of *not valid* can be established, dual to satisfiable (*satisfiable* is of course equivalent to *not unsatisfiable*). The values *valid*, *contingent*, and *not valid* can be defined in terms of *SAT*.

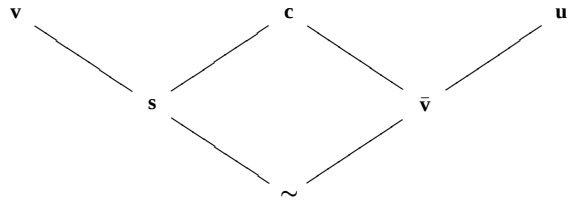
Definition 2 (Valid, Contingent, and Not Valid). Given a formula ϕ, ϕ is *valid* iff $\mathbf{u}\ (\neg\phi)$; ϕ is *contingent* iff both $\mathbf{s}\ (\phi)$ and $\mathbf{s}\ (\neg\phi)$; and ϕ is *not valid* if $\mathbf{s}\ (\neg\phi)$.

The values *valid*, *contingent*, and *not valid* will be denoted by $\mathbf{v}$, $\mathbf{c}$, and $\bar{\mathbf{v}}$, respectively. Further, for a formula ϕ, $\mathbf{v}\ (\phi)$ will denote that ϕ is known to be valid, $\mathbf{c}\ (\phi)$ will denote that ϕ is known to be contingent, and $\bar{\mathbf{v}}\ (\phi)$ will denote that ϕ is known to be not valid.

If knowing the satisfiability of both ϕ and $\neg\phi$ is to have full information regarding the satisfiability of ϕ (and, symmetrically, regarding the satisfiability of $\neg\phi$) and knowing the satisfiability of ϕ but not $\neg\phi$ (or, conversely, $\neg\phi$ but not ϕ) is to have partial information regarding the satisfiability of ϕ, then it can be said that knowing neither the satisfiability of ϕ nor $\neg\phi$ is to have no information regarding the satisfiability of ϕ. If nothing is known about the



(a) Pairs of formula and negation



(b) Satisfiability values

Figure 1: Orderings for (a) pairs of formula and negation and (b) satisfiability values

satisfiability of ϕ or $\neg\phi$ the value of both formulae can be given the value *not known*. The value *not known* will be denoted by $\sim$ and, for a formula ϕ, $\sim(\phi)$ will denote that no value for ϕ or $\neg\phi$ is yet known. Summing up, the set of *satisfiability values* can be defined.

Definition 3 (Satisfiability Values). The set of *satisfiability values* is the set *SatVal* = {**v**, **c**, **u**, **s**, $\bar{\mathbf{v}}$, $\sim$}. The *satisfiability variable* ν ranges over *SatVal*. A *satisfiability claim* for a formula ϕ is an expression $\nu(\phi)$ in which ν is instantiated with one of the values in *SatVal*.

For example, the satisfiability claim **v** (ϕ) is true iff ϕ is known to be valid and the satisfiability claim $\sim(\phi)$ is true iff no satisfiability value for ϕ is known.

The three possibilities with respect to knowing the satisfiability of a formula and its negation, viz., full information, partial information, and no information, provide the basis for a partial ordering of satisfiability values according to what might be called *information content*. The partial ordering of satisfiability value pairs for a formula ϕ and its negation is shown in Figure 1 (a). The more information contained in a pair of values, the higher the pair is in the ordering. For instance, the pair $\langle\sim,\sim\rangle$ represents having no information about the satisfiability of either ϕ or $\neg\phi$, the pair $\langle\sim,\mathbf{s}\rangle$ represents the information that $\neg\phi$ is satisfiable, and the pair $\langle\mathbf{s},\mathbf{s}\rangle$ represents the information that both ϕ or $\neg\phi$ are satisfiable.

Certain possible pairings are obviously omitted, some because they are redundant. The four pairs $\langle\sim,\mathbf{v}\rangle$, $\langle\sim,\mathbf{u}\rangle$, $\langle\mathbf{u},\sim\rangle$ and $\langle\mathbf{v},\sim\rangle$ are omitted because in each case the value $\sim$ can trivially be replaced by **v** or **u** according to the value of the other element in the pair. For example, the $\sim$ in $\langle\sim,\mathbf{v}\rangle$ can immediately be replaced by **u** since if $\neg\phi$ is true for all assignments then ϕ is true for none. Similarly, the two pairs $\langle\mathbf{u},\mathbf{s}\rangle$ and $\langle\mathbf{s},\mathbf{u}\rangle$ are omitted because in each case the value **s** can be replaced by **v**. Finally, the four pairs $\langle\mathbf{v},\mathbf{v}\rangle$, $\langle\mathbf{u},\mathbf{u}\rangle$, $\langle\mathbf{s},\mathbf{v}\rangle$ and $\langle\mathbf{v},\mathbf{s}\rangle$ represent impossible situations. For example, a formula cannot be valid if its negation is satisfiable.

The ordering of Figure 1 (a) can also be represented with respect to the values of *SatVal*, as in Figure 1 (b). A partial ordering is thus defined for *SatVal* .

Definition 4 (Ordering of Satisfiability Values). The set *SatVal* is partially ordered according to information content as follows:

$$\mathbf{v} > \mathbf{s}, \quad \mathbf{c} > \mathbf{s}, \quad \mathbf{c} > \bar{\mathbf{v}}, \quad \mathbf{u} > \bar{\mathbf{v}}, \quad \mathbf{s} > \sim, \quad \bar{\mathbf{v}} > \sim$$

For all values $\nu_1, \nu_2 \in$ *SatVal*, ν_1 is said to be *more precise* (resp. *less precise*) if and only if $\nu_1 > \nu_2$ (resp. $\nu_2 > \nu_1$).

The ordering is assumed to have the usual concept of *least upper bound*, i.e., for two values $\nu_1, \nu_2 \in$ *SatVal*, the least upper bound of ν_1 and ν_2, written $\nu_1 \sqcup \nu_2$, if defined, is the unique value $\nu_3 \in$ *SatVal* such that $\nu_3 \geq \nu_1$ and $\nu_3 \geq \nu_2$ and for all other values $\nu_4 \in$ *SatVal*, if $\nu_4 \geq \nu_1$ and $\nu_4 \geq \nu_2$, then $\nu_4 \geq \nu_3$. For example, $\mathbf{s} \sqcup \sim = \mathbf{s}$, $\mathbf{s} \sqcup \bar{\mathbf{v}} = \mathbf{c}$ and $\mathbf{v} \sqcup \mathbf{v} = \mathbf{v}$, but $\mathbf{v} \sqcup \mathbf{c}$, $\mathbf{v} \sqcup \mathbf{u}$, and $\mathbf{c} \sqcup \mathbf{u}$ do not exist.

4. OBTAINING SATISFIABILITY VALUES

It was noted that the single query *SAT* (ϕ) decides only whether **s** (ϕ) or **u** (ϕ). However, in certain cases satisfiability values other than **s** and **u** can be obtained through inference. For example, if the result **s** (ϕ) has been previously established, then a new result **s** $(\neg\phi)$ would allow both **c** (ϕ) and **c** $(\neg\phi)$ to be inferred. Or, if the new result is **u** $(\neg\phi)$, then **v** (ϕ) can be inferred. A full algebra defining the possible inferences for the values in *SatVal* is documented in [10]. The patterns of analysis described in the next section make use of a *lookup table* that records satisfiability values for formulae as they are discovered during analysis. The lookup table is a map from formulae to satisfiability values, obtained either by satisfiability queries or inferred from previous results. Initially, all formulae are mapped to the value $\sim$.

Definition 5 (Satisfiability Lookup Table). A *satisfiability lookup table* is a map $SAT_{Table} : \Phi \rightarrow$ *SatVal* from formulae to satisfiability values.

The value of a formula ϕ can now be obtained by querying either the satisfiability oracle or the lookup table. Either way, it is desirable to obtain the more precise value. For example, when querying the satisfiability value of ϕ, if $SAT(\phi) = \mathbf{s}$ but $SAT_{Table}(\phi) = \mathbf{v}$ (which would be the case when the more precise value **v** had been previously inferred for ϕ), then the value **v** should be taken, since $\mathbf{v} > \mathbf{s}$. Given two satisfiability values, ν_1 and ν_2, the most precise value obtainable is the least upper bound of the two, i.e., $\nu_1 \sqcup \nu_2$. Note that the most precise value need not in fact be either ν_1 or ν_2. For example, if $\nu_1 = \mathbf{s}$ and $\nu_2 = \bar{\mathbf{v}}$, then the least upper bound, and therefore most precise value obtainable, is **c**. A lookup table is sound in the sense that the least upper bound always exists for a given update. The following function gives the value of a formula.

Definition 6(Obtaining the Satisfiability Value of a Formula). Given an oracle *SAT* and satisfiability lookup table SAT_{Table}, the function

$$GetVal : \Phi \rightarrow SatVal$$

gives a satisfiability value for a formula, such that, for all $\phi \in \Phi$

$$GetVal(\phi) = SAT(\phi) \sqcup SAT_{Table}(\phi).$$

However, as new values are learnt for a formula ϕ, a lookup table may need to be updated so that the value obtained through *GetVal* is always the most precise value yet discovered in a given analysis. Occasionally, a value may be inferred for a formula during analysis that is less precise than that already recorded in the lookup table. For example, if a conjunction $\phi_1 \wedge \phi_2$ is found to be satisfiable, it is implied that the conjuncts ϕ_1 and ϕ_2 are also satisfiable. However,

if lookup table already maps ϕ_1 to **v**, then this entry should not be updated with the value **s**. The lookup table mapping should never be updated with a value for ϕ that is less precise than its existing value. An appropriate update function is defined below.

Definition 7(Lookup Table Update). For a formula ϕ, satisfiability value $\nu \in$ *SatVal*, and lookup table SAT_{Table}, an *updated lookup table* SAT'_{Table} is given by the function $Upd\ (SAT_{Table}, \phi, \nu)$ such that

$$Upd\ (SAT_{Table}, \phi, \nu) = SAT_{Table} \oplus \phi \mapsto \nu$$

iff $\nu > SAT_{Table}\ (\phi)$. Otherwise $Upd\ (SAT_{Table}, \phi, \nu) = SAT_{Table}$.

5. PATTERNS

Application of the analysis patterns starts with the analysis pattern for the top-level formula being queried. As analysis moves to the subformulae of the formula in question the patterns are applied recursively according to the top-level connective of whatever subformula is currently being analysed. Each satisfiability query for a formula ϕ is represented by $?\ (\phi)$. Each satisfiability query $?\ (\phi)$ is resolved by a corresponding call to $GetVal\ (\phi)$, the result of which determines the next formula to be analysed as dictated by the patterns. The progression of satisfiability queries from formula to formula is not necessarily a linear sequence. It is often the case that analysis of a formula branches into parallel analyses of its subformulae.

There is an analysis pattern for each of the propositional connectives common to most popular specification languages (e.g. JML, Spec♯, Alloy): *Negation Pattern*, *Conjunction Pattern*, *Disjunction Pattern*, and *Implication Pattern*. While it may be possible to consider only a basic set of patterns in which implication and either conjunction or disjunction are reducible to the remaining connectives, the full set is presented here for clarity. Analysis of an implication in particular is less straightforwardly presented without its corresponding pattern.

Analysis of a given formula is guided step by step by repeated application of the patterns, beginning with the pattern that matches the root connective of that formula. For example, the pattern matching formulae of the form $\phi_1 \Rightarrow \phi_2$ dictates that ϕ_1 and $\neg\phi_2$ should be checked for satisfiability if $\phi_1 \Rightarrow \phi_2$ is valid. Where application of a pattern identifies that a formula is valid or unsatisfiable due to a valid or unsatisfiable subformula, analysis terminates with a warning accompanied by a reference to the subformulae in which a potential error may reside. There is also a pattern for atomic (connective-free) formulae known as the *Base Pattern*. Application of the Base Pattern to a contingent atomic formula results in default (i.e., *warning-free*) termination of a decompositional analysis process.

The patterns are represented as the decision diagrams shown in Figures 3–7. In these diagrams, a non-terminal node represents a satisfiability query, e.g., $?\ (\phi)$, and, for two nodes *A* and *B*, a transition from *A* to *B* represents an application of $GetVal\ (\phi)$, where ϕ is the formula contained in the satisfiability query of *A*. Transitions are labelled with a satisfiability value in *SatVal*. A transition labelled with satisfiability value ν is taken from a node representing the satisfiability query $?\ (\phi)$ iff $GetVal\ (\phi) = \nu$. For example, a transition labelled **v** is taken from a node representing the satisfiability query $?\ (\phi)$ iff $GetVal\ (\phi) = \mathbf{v}$. Where there are multiple transitions from node *A* to node *B*, a single transition is shown but with multiple labels.

The notation used in the diagrams of Figures 4–7 is summarised in Figure 2. Non-terminal nodes are depicted as shown in Figure 2 (a). The start node of each pattern is known as the *pattern root* and depicted as shown in Figure 2 (b). A pattern root represents a satisfiability claim for a formula whose top-level connective is matched by the pattern. For instance, the root of the Implication Pattern (Figure 7) contains the satisfiability claim $?\ (\phi_1 \Rightarrow \phi_2)$. A pattern is said to be *applied* to the formula in its root and the formula in the root is known as the *root formula* of the pattern.

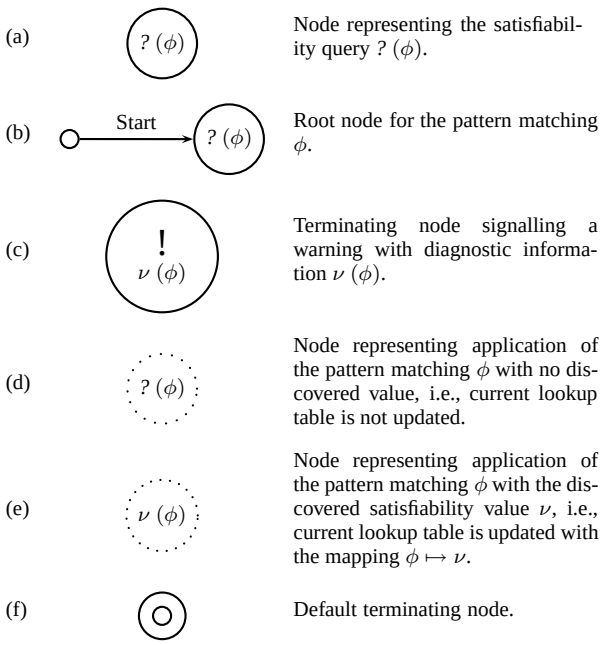


Figure 2: Summary of Pattern Notation

A terminal node in a pattern is known as a *pattern leaf*. There are three kinds of pattern leaf:

- Leaf signalling a warning
- Leaf representing an application of a further pattern
- End leaf

A leaf that signals a warning is depicted as shown in Figure 2 (c). If application of a pattern terminates with a warning, it signals a potential specification error. A warning is issued in the following cases:

- An atomic formula is valid
- An atomic formula is unsatisfiable
- An implication is valid due to an unsatisfiable antecedent
- An implication is valid due to a valid consequent

For example, if an atomic formula contains a pure method term it will be unsatisfiable if the pure method term is undefined for all program assignments. Further, an atomic formula can be valid if it contains a pure method term whose precondition and postcondition are valid. A warning node also contains diagnostic information in the form of a satisfiability claim. For example, in the application of an Implication Pattern to a formula $\phi_1 \Rightarrow \phi_2$, a warning may be issued with the satisfiability claim $\mathbf{u}\ (\phi_1)$ indicating that the root formula is possibly valid due to an error in the specification of ϕ.

A leaf that represents an application of a further pattern is depicted as shown in Figure 2 (d)-(e). Such a termination of a pattern means that the pattern for the formula ϕ should be applied to investigate further. The node may contain a satisfiability query, as in Figure 2 (d), indicating that the satisfiability value of ϕ may not be known. However, in some cases, application of the pattern leading

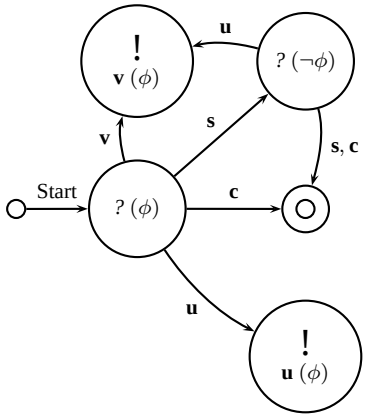


Figure 3: Base Pattern

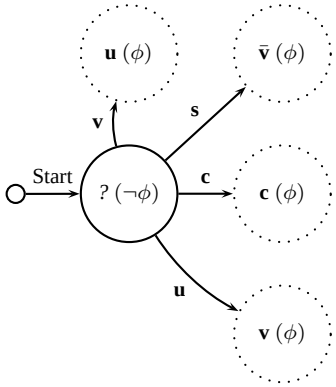


Figure 4: Negation Pattern

to this leaf node may have provided a value for ϕ. In this case, the leaf node may contain a satisfiability claim, e.g., $\mathbf{v}\ (\phi)$, as in Figure 2 (e). This indicates that the given satisfiability value, e.g., **v**, is entered into the satisfiability lookup table for ϕ, i.e., a new mapping $SAT_{Table}' = SAT_{Table} \oplus \phi \mapsto \mathbf{v}$ is constructed. In this way, information about the satisfiability of formula is accumulated through recursive application of the patterns. Finally, an end leaf is depicted as shown in Figure 2 (f). An end leaf appears only once, in the representation of the Base Pattern, and represents default termination of an application of the patterns.

Base Pattern. The Base Pattern (Figure 3) is applied to atomic formulae. If an atomic formula ϕ is unsatisfiable (i.e., $GetVal\ (\phi) = \mathbf{u}$), the transition labelled **u** is taken from the pattern root and a warning is issued signalling the unsatisfiability of ϕ. Similarly, if ϕ is known to be valid (i.e., $GetVal\ (\phi) = \mathbf{v}$), the transition labelled **v** is taken and a warning is issued signalling the validity of ϕ. If ϕ is known to be contingent (i.e., $GetVal\ (t) = \mathbf{c}$), the transition labelled **c** is taken and application of the pattern ends normally since a term being true in some states and false in others is as expected. Otherwise, ϕ is only known to be satisfiable (i.e., $GetVal\ (t) = \mathbf{s}$), and the transition labelled **s** is taken. In this case, the negation of ϕ is then checked for satisfiability to determine whether or not ϕ is valid. If $\neg\phi$ is satisfiable (or known to be contingent, as indicated by the label **s**, **c**), then ϕ is contingent and no warning need be issued: application of the pattern terminates normally. But if $\neg\phi$ is unsatisfiable, then ϕ is valid and, as above, a warning signalling the validity of ϕ is again issued. Note that the check for the satisfiability of $\neg\phi$ is included in the Base Pattern and not treated as an application of the Negation Pattern. This is to avoid cycling between the Base Pattern, which given ϕ checks $\neg\phi$, and the Negation Pattern, which given $\neg\phi$ checks ϕ.

Negation Pattern. The Negation Pattern (Figure 4) is applied to formulae of the form $\neg\phi$. Application of this pattern permits the exploration of whether or not $\neg\phi$ is valid or unsatisfiable and hence, conversely, whether or not ϕ is unsatisfiable or valid. If a formula $\neg\phi$ is unsatisfiable (i.e., $GetVal\ (\neg\phi) = \mathbf{u}$), the transition labelled **u** is taken from the pattern root. Since ϕ has been discovered to be valid, the pattern matching the formula ϕ is applied to investigate further. If $\neg\phi$ is known to be valid (i.e., $GetVal\ (\neg\phi) = \mathbf{v}$), the transition labelled **v** is taken and since ϕ is discovered to be unsatisfiable, the pattern matching the formula ϕ is applied to investigate further. If $\neg\phi$ is known to be contingent (i.e., $GetVal\ (\neg\phi) = \mathbf{c}$), then ϕ must be contingent. Though a formula ϕ is expected to be contingent, valid and unsatisfiable subformulae can still hide

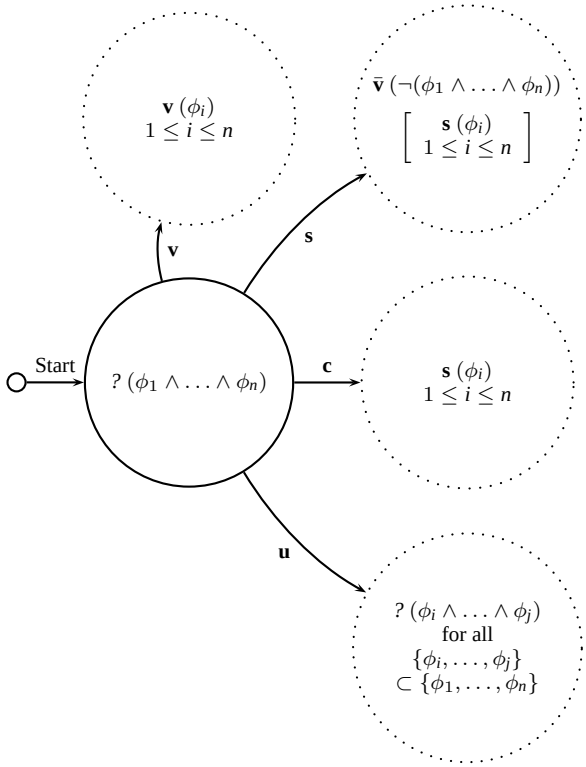


Figure 5: Conjunction Pattern

beneath a contingent formula, so ϕ may be investigated further through application of its pattern. Otherwise, $\neg\phi$ is known only to be satisfiable (i.e., $GetVal\ (\neg\phi) = \mathbf{s}$), implying that ϕ cannot be valid (though possibly unsatisfiable), i.e., $\bar{\mathbf{v}}\ (\phi)$. ϕ can again be further investigated through application of its pattern. Note that the Negation Pattern does not check the negation of its root formula since analysing $\neg\neg\phi$ is of course equivalent to analysing ϕ, which would be redundant given that ϕ is always next analysed through application of its pattern.

Conjunction Pattern. The Conjunction Pattern (Figure 5) is applied to formulae of the form $\phi_1 \wedge \ldots \wedge \phi_n$. Application of this pattern decomposes a conjunction into its conjuncts to identify whether the conjunction is valid or, if it is unsatisfiable, which con-

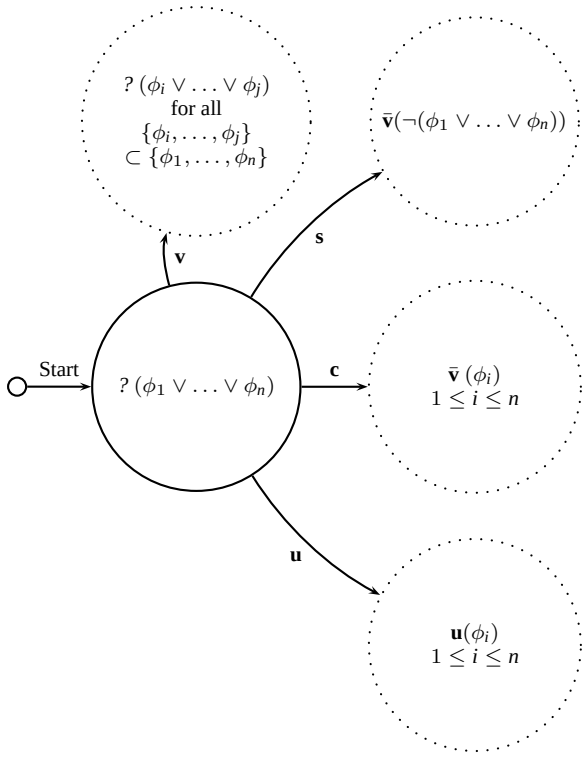


Figure 6: Disjunction Pattern

juncts or combination of conjuncts are unsatisfiable. If $\phi_1 \wedge \ldots \wedge \phi_n$ is unsatisfiable (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{u}$), the transition labelled **u** is taken from the pattern root. However, it is not necessarily the case that any single conjunct is unsatisfiable. Investigating further by simply applying the pattern for each conjunct is insufficient since a conjunction can be unsatisfiable even though each conjunct is satisfiable in isolation. This is because two conjuncts ϕ_i and ϕ_j, say, may be inconsistent. In other words, ϕ_i may imply $\neg\phi_j$ and vice versa. To investigate an unsatisfiable conjunction further, the patterns for all arbitrary subsets of conjunct are applied. In this case, no value for each combination is yet implied.

On the other hand, if $\phi_1 \wedge \ldots \wedge \phi_n$ is known to be valid (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{v}$), the transition labelled **v** is taken. This transition implies that each conjunct ϕ_i is valid and the validity of ϕ_i, for $1 \leq i \leq n$ is investigated through application of its pattern. If $\phi_1 \wedge \ldots \wedge \phi_n$ is known to be contingent (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{c}$), the transition labelled **c** is taken. For the conjunction to be contingent, no conjunct can be unsatisfiable, i.e., either $\mathbf{v}$ (ϕ) or $\mathbf{c}$ (ϕ) for each ϕ_i. Moreover, since it is known that the conjunction is not valid, at least one conjunct (and possibly all) must be contingent. However, which conjuncts are contingent and which valid, if any, cannot be determined without further application of the patterns for each ϕ_i. The most that can be stated at this point about each ϕ_i is that it is not unsatisfiable, i.e., $\mathbf{s}$ (ϕ_i).

Finally, if $\phi_1 \wedge \ldots \wedge \phi_n$ is known only to be satisfiable (i.e., *GetVal* $(\phi_1 \wedge \ldots \wedge \phi_n) = \mathbf{s}$), the pattern for its negation is applied to determine whether or not $\phi_1 \wedge \ldots \wedge \phi_n$ is valid. The lookup table can here be augmented in two ways. First, with the value $\bar{\mathbf{v}}$ (not valid) for $\neg(\phi_1 \wedge \ldots \wedge \phi_n)$ since the root formula is known to be satisfiable, and second, with the value **s** for each ϕ_i since, if the conjunction is satisfiable so are each of its conjuncts. In Figure 5, this second update is shown in parenthesis.

Disjunction Pattern. The Disjunction Pattern (Figure 6) is applied to formulae of the form $\phi_1 \vee \ldots \vee \phi_n$. Application of this pattern decomposes a disjunction into its disjuncts to identify whether the disjunction is valid or unsatisfiable, and if valid, which disjuncts or combination of disjuncts are valid. The Disjunction Pattern is the dual of the Conjunction Pattern in that the valid (resp. unsatisfiable) case of the Disjunction Pattern is the dual of the unsatisfiable (resp. valid) case of the Conjunction Pattern. Considering each case in turn, if $\phi_1 \vee \ldots \vee \phi_n$ is *unsatisfiable* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{u}$) the transition labelled **u** is taken from the pattern root. For the disjunction to be unsatisfiable all disjuncts must be unsatisfiable and the pattern for each ϕ_i, $1 \leq i \leq n$ is applied to investigate further. The lookup table is updated to map each ϕ_i to the value **u**.

If $\phi_1 \vee \ldots \vee \phi_n$ is known to be *valid* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{v}$) the transition labelled **v** is taken. Note that $\phi_1 \vee \ldots \vee \phi_n$ may be valid not only on account of a valid disjunct, but because two disjuncts ϕ_i and ϕ_j, say, may be inconsistent such that ϕ_i implies $\neg\phi_j$ and vice versa. Therefore, to investigate a valid disjunction further, the patterns for all arbitrary subsets of disjunct are applied. No value for each combination is implied so the lookup table is not updated for any $\phi_i \vee \ldots \vee \phi_j$.

If $\phi_1 \vee \ldots \vee \phi_n$ is known to be *contingent* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{c}$), the transition labelled **c** is taken. Since the disjunction is not valid, no disjunct can be valid (i.e., $\bar{\mathbf{v}}$ (ϕ_i), for $1 \leq i \leq j \leq n$) and the lookup table is updated with the value $\bar{\mathbf{v}}$ for each ϕ_i before applying the pattern for each ϕ_i to investigate further. Finally, if it is known only that $\phi_1 \vee \ldots \vee \phi_n$ is *satisfiable* (i.e., *GetVal* $(\phi_1 \vee \ldots \vee \phi_n) = \mathbf{s}$) and the transition labelled **s** is taken. To determine whether or not the root formula is valid the pattern for $\neg(\phi_1 \vee \ldots \vee \phi_n)$ is applied. Again, in this case, it is known that no disjunct can be valid and the lookup table is updated to map each ϕ_i to $\bar{\mathbf{v}}$. Note that this case is the dual of the unsatisfiable case in the Conjunction Pattern.

Implication Pattern. The Implication Pattern (Figure 7) is applied to formulae of the form $\phi_1 \Rightarrow \phi_2$. Application of this pattern investigates whether its root formula is valid because of an unsatisfiable antecedent or valid consequent or, if the root formula is unsatisfiable, why the antecedent is valid and consequent unsatisfiable. If $\phi_1 \Rightarrow \phi_2$ is *unsatisfiable* (i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{u}$), *both* transitions labelled **u** are taken from the pattern root. For the implication to be unsatisfiable, ϕ_1 must be valid and ϕ_2 must be unsatisfiable. The validity of ϕ_1 and the unsatisfiability of ϕ_2 can be investigated further through application of the respective patterns for each.

If $\phi_1 \Rightarrow \phi_2$ is known to be *valid* (i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{v}$), both transitions labelled **v** are taken. The validity of the root formula must be due either to the validity of ϕ_1 or the unsatisfiability of ϕ_2 (or both). On one branch, the satisfiability of ϕ_1 is checked. If ϕ_1 is unsatisfiable, a warning is issued indicating that $\phi_1 \Rightarrow \phi_2$ is valid on account of an unsatisfiable antecedent. Otherwise, if ϕ_1 is found to be any of valid, contingent or satisfiable (i.e., *GetVal* $(\phi_1) \in \{\mathbf{s}, \mathbf{v}, \mathbf{c}\}$), the formula can be investigated further through application of its pattern. If new values for ϕ_2 and $\neg\phi$ are known, the lookup table is updated. On the second branch labelled **v**, the satisfiability of $\neg\phi_2$ is checked. If $\neg\phi_2$ is unsatisfiable, a warning is issued indicating that $\phi_1 \Rightarrow \phi_2$ is valid on account of a valid consequent. Otherwise, if $\neg\phi_2$ is found to be any of any of valid, contingent or satisfiable, i.e., *GetVal* $(\phi_1) \in \{\mathbf{s}, val, \mathbf{c}\}$, the formula ϕ_2 is not valid and it can be investigated further through application of its pattern. Again, if new values for ϕ_2 and $\neg\phi$ are known, the lookup table is updated.

On the other hand, if $\phi_1 \Rightarrow \phi_2$ is known to be *contingent*, i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{c}$, the transition labelled **c** is taken from

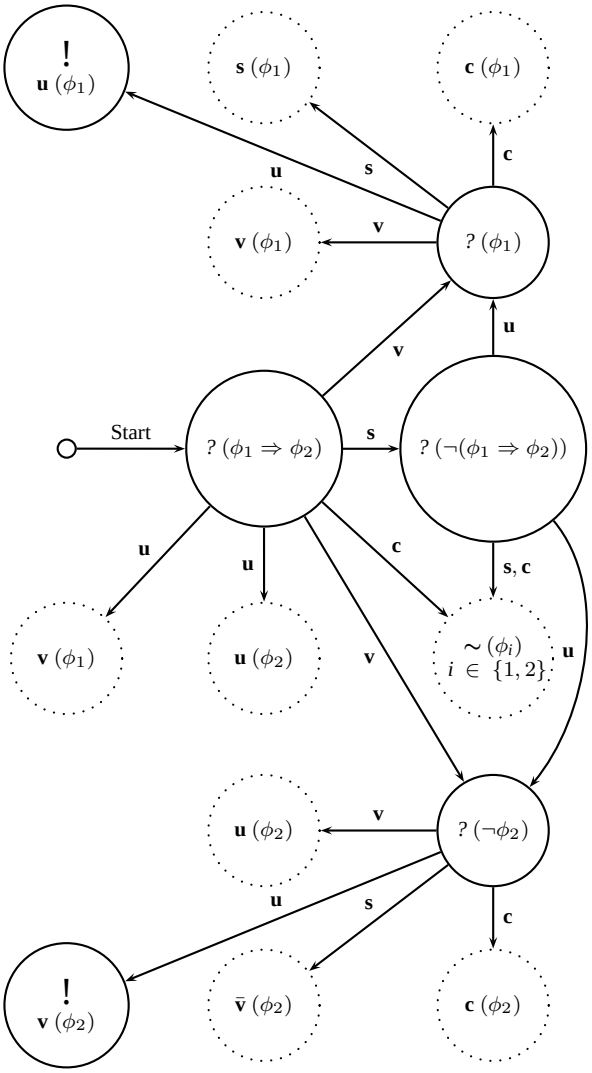


Figure 7: Implication Pattern

the pattern root. In this case, nothing is known about the values of ϕ_1 and ϕ_2, further investigation may be carried out through application of the patterns for these formulae. Finally, if $\phi_1 \Rightarrow \phi_2$ is known only to be *satisfiable*, i.e., *GetVal* $(\phi_1 \Rightarrow \phi_2) = \mathbf{s}$, the transition labelled **s** is taken and the satisfiability of the negation of the root formula is checked. If the negation is satisfiable, the root formula and its negation are contingent and the patterns for ϕ_1 and ϕ_2 may be applied. If the negation is unsatisfiable, then $\phi_1 \Rightarrow \phi_2$ is discovered to be valid and the steps described above are taken.

6. IMPLEMENTATION AND EVALUATION

A prototype tool has been developed to evaluate the effectiveness of the patterns in practice. The tool automatically analyses formulae of a specification language called *Loy*, which is essentially a subset of JML. Details of Loy and the tool are given in [10]. The tool harnesses the SAT-solving capabilities of the Alloy Analyzer by encoding Loy specifications into the Alloy input language and then running satisfiability queries on this encoding. The results of this analysis are then fed back to the user in terms of the original Loy specification.

Returning to the example of Section 2, it is now possible to illustrate how an automated pattern-driven analysis can uncover the hidden inconsistency introduced during the evolution of the method specification. For clarity, some of JML's ascii syntax is ignored in the following (e.g. $\Rightarrow$ is written for ==> and $\wedge$ for &&). The top-level formula to be analysed is the implication to test that the postcondition of *insert* in *BoundedQueue* implies that of *insert* in *Queue*. The Implication Pattern is applied to this formula and the first satisfiability query finds it to be satisfiable. The branch labelled **s** is taken from the start node and so the satisfiability of the formula's negation is queried next. The negation is found to be unsatisfiable. At this point, the Implication Patten splits and analysis of the antecedent and consequent continue in parallel. Satisfiability of the antecedent is queried and it is found to be unsatisfiable:

u *((result* $\Rightarrow$ *contains(e))* $\wedge$ *entries = \old(entries.add(e)*
$\wedge$ *size()* $<$ *\old(entries.size())* $\wedge$ *size()* $\leq$ *MAX))*

Here, the tool issues a warning to the user, with the information that the original formula is valid, or vacuously satisfiable, on account of its inconsistent antecedent. Along the other branch, satisfiability of the negation of the consequent is queried, which is found to be satisfiable:

s *((result* $\Rightarrow$ *contains(e))* $\wedge$ *entries = \old(entries.add(e)*

At this point, the Conjunction Pattern can be applied to both the antecedent and the consequent to investigate the inconsistency of the former and to check whether the latter is vacuously satisfiable or contingent. In the prototype tool, automated analysis pauses here and the user prompted for continuation. It is likely that a user would be more interested in pursuing analysis of the inconsistent antecedent. Automated application of the Conjunction Pattern finds the inconsistent formulae to be *size()* $<$ *\old(entries.size())* and *entries = \old(entries.add(e))* (part of this step is the application of the Base Pattern to ascertain that these formulae are not internally inconsistent, i.e., that the method specifications of *size()* or *add()* are not individually inconsistent). Further automated applications of the Conjunction Pattern to the formulae in the specifications of *size()* and *add()* identifies the inconsistency to be the following:

entries.size() $>$ *\old(entries.size())* $\wedge$ *size()* $<$ *\old(entries.size())*
$\wedge$ *size() == entries.size())*

The user should hopefully now see that the inconsistency is due to a typo in the evolved specification of *insert()*: the new postcondition from Section 2 should read

size() $>$ *\old(entries.size()) && size()* $<=$ *MAX*

with a $>$ instead of $<$ in the first conjunct.

The soundness and completeness of pattern-driven analysis in practice of course depends on the soundness and completeness of the SAT-solver used in a given implementation. The Alloy Analyser is sound but not complete (the tool can run out of memory while undertaking certain queries). Practical considerations aside, pattern-driven analysis can be considered complete insofar as it terminates and finds the satisfiability values for any propositional formula and its subformulae. It will also issue a warning for any valid or unsatisfiable formula. Further, pattern-driven analysis can be considered sound insofar as the satisfiability values found for any propositional formula and its subformulae are correct, since these will come either from SAT or a sound inference from previous re-

sults [10].

Finally, the complexity of a pattern-driven analysis can be exponential. The worst case is the Conjunction Pattern, which may direct analysis through an exhaustive querying of each combination of conjunct in the root formula to determine a source of inconsistency. However, application of the Conjunction Pattern does not always take this branch and only does so when an inconsistency is present.

7. RELATED WORK

The concept of patterns to support an activity is of course not new. Design patterns [7] now provide handy templates for programmers across the world. Even in the area of formal specification, the idea of providing patterns to support specifiers has been considered before [6]. However, the patterns presented here are not patterns to be used by programmers or specifiers but the foundation of an automated analysis framework. They are templates for sound decision procedures whose implementation can be hidden from users.

Recent work on helping users of specification analysis tools avoid being misled by feedback includes [4] and [13], both dealing with JML. [4] presents an extension of an existing JML static checking tool which warns a user whenever a term in a precondition formula is undefined, thus avoiding cases where a method is spuriously reported to meet its specification simply because its precondition is unsatisfiable. The analysis patterns presented in this paper address similar concerns but are far more general in their application, since they check the satisfiability of arbitrary (propositional) formulae rather than focus on a particular cause of unsatisfiability such as undefinedness. [13] suggests including unsoundness and incompleteness disclaimers in feedback from automated analysis tools so that users are less likely to have misplaced confidence in feedback from unsound or incomplete analyses. Similar in spirit to the use of analysis patterns, this approach would be complementary to a patterns-based analysis toolset.

The notion of checking for vacuity is not new. For example, [2] addresses a form of vacuity checking in work on hardware verification. In the broader software verification setting, work also exists on catching vacuity in temporal model checking [14, 9].

Finally, the work in this paper is a greatly extended version of that published in [11]. This earlier work introduced the idea of analysis patterns but considered only cases of satisfiability and unsatisfiability. A contribution of the present paper is the definition of a lattice of six satisfiability values, which are used during pattern application. The feedback available from an application of the new analysis patterns therefore exceeds that provided by the previous versions.

8. CONCLUSION

This paper presented a set of automatable analysis patterns that constitute a framework for automated analysis of software specifications. The difficulties faced by developers and maintainers of a specification could be greatly reduced by an automated specification management environment. The analysis patterns and their prototype implementation are a step towards the provision of such an environment. In particular, the analysis patterns allow many hidden logical errors to be uncovered automatically, without need for expert direction and without offering misleading feedback. Due to a lack of space, two patterns for first order formulae (the *Universal-* and the *Existential Quantification Pattern* [10]), which primarily catch errors due to empty domains, have not been discussed here.

Further work will involve evaluating the approach with a full specification language such as JML, perhaps by extending an existing toolset. This would allow the scalability of the approach to be explored. Work in [10] on inference of satisfiability values from previous results could also be extended to provide further support for pattern-driven analysis. The more values that can be inferred, the fewer calls to a SAT-solver and the quicker an analysis. In addition, a non-trivial case study must be undertaken to investigate the limitations of the approach in application to industrial-scale specifications in practice.

Acknowledgements The authors wish to thank Michael Huth for his many comments on the work in this paper.

REFERENCES

[1] M Barnett, R Leino, and W Schulte. The Spec♯ Programming System: An Overview. *CASSIS*, 2004.

[2] Derek L. Beatty and Randal E. Bryant. Formally Verifying a Microprocessor Using a Simulation Methodology. *DAC*, 1994.

[3] L Burdy, Y Cheon, D Cok, M Ernst, J Kiniry, G Leavens, K Leino, and E Poll. An Overview of JML Tools and Applications. *STTT*, 2005.

[4] P Chalin. Early Detection of JML Specification Errors Using ESC/Java2. *SAVCBS*, 2006.

[5] K Dhara and G Leavens. Forcing Behavioral Subtyping Through Specifcation Inheritance. *ICSE*, 1996.

[6] M Dwyer, G Avrunin, and J Corbett. Property Specification Patterns for Finite-State Verification. *FMSP*, 1998.

[7] E Gamma, R Helm, R Johnson, and J Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *Lecture Notes in Computer Science*, 707, 1993.

[8] J Gu, P Purdom, J Franco, and B Wah. Algorithms for the Satisfiability (SAT) Problem: a Survey. *Satisfiability Problem: Theory and Applications*, 1997.

[9] Arie Gurfinkel and Marsha Chechik. Extending Extended Vacuity. *FMCAD*, 2004.

[10] W Heaven. *Object-Oriented Specification: Analysable Patterns and Change Management*. PhD Thesis, Dept. of Computing, Imperial College London, 2007.

[11] W Heaven and A Russo. Enhancing the Alloy Analyzer with Patterns of Analysis. *WLPE*, 2005.

[12] D Jackson. *Software Abstractions*. The MIT Press, 2006.

[13] J Kiniry, A Morkan, and B Denby. Soundness and Completeness Warnings in ESC/Java. *SAVCBS*, 2006.

[14] Orna Kupferman and Moshe Y. Vardi. Vacuity Detection in Temporal Model Checking. *STTT*, 1999.

[15] G Leavens, A Baker, and C Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *Software Engineering Notes*, 31(3), 2006.

[16] LIFC. JML-Testing-Tools. http://lifc.univ-fcomte.fr/jmltt/.

[17] B Liskov and J Wing. A Behavioral Notion of Subtyping. *TOPLAS*, 1994.

[18] M Prasad, A Biere, and A Gupta. A Survey of Recent Advances in SAT-Based Formal Verification. *STTT*, 7(2), 2005.

Extensions of the theory of observational purity and a practical design for JML

David R. Cok
Eastman Kodak Company
Research Laboratories
1999 Lake Avenue
Rochester, NY 14650 USA
david.cok@kodak.com

Gary T. Leavens
School of Electrical Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA
leavens@eecs.ucf.edu

ABSTRACT

To prevent erratic behavior during runtime checking, JML only allows assertions to call pure, i.e., side-effect free, methods. However, JML's notion of purity checking is too conservative. For example, `Object`'s `equals` method needs to be used in assertions, but some classes use side effects in their `equals` method to maintain hidden caches or to trigger lazy evaluation, and so these methods cannot be pure in JML's sense. To handle such cases JML and similar interface specification languages need a less conservative notion of pure methods. In this paper we apply and slightly extend the existing theory of "observationally pure" methods to JML, and explain our language design. This design is practical and accommodates common uses. Our extension of current theory provides appropriate encapsulation combined with inheritance, invariants, method specifications, frame conditions, secret helper methods, and multiple sets of secret state locations. We also introduce a semantics for static analysis that preserves correctness without imposing non-interference.

Keywords: specification languages, runtime assertion checking, documentation, tools, observational purity, query method, pure method, formal methods, program verification, programming by contract, Java language, JML language, OpenJML, OpenJDK.

1. INTRODUCTION

1.1 Motivation

Subroutines, or methods, are a useful and important abstraction mechanism in both programming and in specification. Using the methods of an object-oriented programming language in design-by-contract style specification languages (such as Eiffel [13], JML [10, 8], and Spec# [3]) avoids duplication between program code and specification and improves understanding. For example, specifications often use methods that extract values from objects (e.g., `getX`) and that compare the values of objects (e.g., `equals` in Java).

However, using program methods in specifications causes several potential problems. Operationally, it is important to prevent runtime assertion checking from observably changing the results of a program, as side effects from assertion checking would greatly complicate debugging [13, 8]. Mathematically, formal verification is easiest when specifications are simple and direct, and thus do not involve feedback between the meaning of the specification and the program being specified. Thus, for both dynamic and static checking, methods that have no effect on a program's state are most suitable for use in specification, as they map neatly to simple mathematical abstractions. For these reasons JML only allows "pure" Java methods to be used in assertions.

However, our specification experience is that JML's current restrictions cause serious problems. One problem is `Object`'s `equals` method in Java. This method is key to the specification of Java's `Collection` subtypes, since programs use it to decide when elements are equivalent. Thus specifications need to call this method, and hence in JML it must be pure. But, due to JML's specification inheritance [7], specifying `Object`'s `equals` method as pure means that in all subtypes of `Object`, i.e., in all types, it must be pure. However, there are several examples of `equals` methods in Java libraries and applications that are not pure [10].

Thus what is needed is a weaker notion of purity that allows such examples but still guarantees safe runtime checking and allows for simple mathematical modeling.

1.2 Observational Purity

Several researchers [2, 4, 14] have noted this problem and investigated the theory of *observationally pure* methods, which have limited and well-encapsulated effects on a program's state. The state changes made by such methods are hidden from the bulk of the program, and thus such methods can be used safely in specifications. However, verifying that such changes do not cause interference with other computations in the program is difficult. To date there is no interface specification language with a practical and implemented design allowing observationally pure methods to be called in specifications.[1] This lack is a serious impediment to writing specifications on a large scale (e.g., for the JDK libraries) and to progress in using static verification on real-world software.

Observational purity is one of a number of levels of purity that have been identified in past discussions [1]: *strong purity* – no side effects (besides consumption of time, acquiring and release of stack space); *weak purity* – allocation and modification of new objects allowed; and *observational purity* – confined changes to the program state allowed. JML's `pure` annotation corresponds to weak purity. Observational purity is the focus of this paper, hence for brevity we

[1]There have been some beginnings of implementation in Spec# [3] and by Cok in JML.

```
class Super {
  //@ protected normal_behavior
  //@   ensures \result >= 0;
  protected /*@ spec_public pure @*/ int computeValue() {
    int i = 0; /* ... expensive code ... */
    return i;
  }

  //@ public normal_behavior
  //@   ensures \result == computeValue();
  public /*@ query @*/ int getValue() {
    return computeValue();
} }

class Cache extends Super {
  /*@ secret*/ private int cachedValue; //@ in getValue;
  /*@ secret*/ private boolean isCached = false;
                                        //@ in getValue;

  /*@ secret private invariant isCached ==>
                     (cachedValue == computeValue()); */

  public /*@ query */ int getValue() {
    if (!isCached) {
      cachedValue = computeValue();
      isCached = true;
    }
    return cachedValue;
  }

  @Secret("getValue")
  public /*@ pure @*/ boolean isCached() {
    return isCached;
} }
```

Figure 1: Caching a value that is expensive to compute.

will often shorten "observationally pure" and "observational purity" to "*opure*" and "*opurity*."

The classic example (discussed by others also, e.g., [2, 9]) of an opure method uses a cache, as shown in Fig. 1 (ignore the JML annotations, contained in comments, for now). In the figure, method `getValue()` computes some value; the computation, performed by `computeValue()`, is expensive and may be needed more than once. In the superclass, `computeValue()` is called each time. In the subclass, when the method is first called the computation is performed and the result stored in a private location; on subsequent calls, the stored value is returned. If not for the assignment to the private field `cachedValue`, the subclass's `getValue()` method would be weakly pure and thus readily used in a specification. To complicate matters, the superclass's `getValue()` method is weakly pure, but a call to `getValue()` may invoke the subclass's method, which has side effects.

Despite these complications, `getValue()` is opure, and so could be used in a specification, since the locations it modifies are neatly encapsulated within class `Cache`, and it seems that they have no influence on the execution of a calling program. But we need mechanisms to guarantee that the alteration of program state does not change the result of any future computations, such as through the use of `isCached()`.

A more complex example is a shared database. For example, one could cache the results of several methods in a shared database that maps tuples of arguments to computed results. In this case, we need to guarantee both that storing one result does not affect others and that knowledge of whether a result is stored does not inappropriately affect the program's execution. Here, the information intended to be secret is not so neatly encapsulated.

1.3 Goals and Problem

Our main goal is to allow some use of opure methods in specifications. This usage must allow safe runtime assertion checking and have a simple and consistent semantics for static verification. That is, runtime assertion checking must guarantee that the side effects of executing a specification do not observably affect the execution of the program. Furthermore, the semantics of static verification must be usable and consistent with program executions that perform assertion checks during runtime.

We would like to preserve other goals of JML [8] as well. To the extent possible we want to continue to use JML to specify Java programs as they are written, without constraining valid programs to some Java subset. In particular, we prefer not to require a specific use of Java visibility modifiers in order to accommodate opurity.

1.4 Contributions

In brief, our solution follows previous work [2, 4, 14] by allowing opure methods to be used in specifications; the keyword **`query`** declares such methods. The portion of the program state that a query method may modify is declared using **`secret`** and is called *secret* state, since we wish it to be unobserved by the remainder of the program [4]. The remainder of the program state is *open*.

This paper makes the following contributions toward solving the practical problems of observational purity:

- we propose a specific application of current theory to a language design in JML;
- in the process we identify two issues with the current theory;
- we extend the current theory to accommodate multiple pieces of secret state, inheritance, invariants and method specifications, frame conditions, and secret helper methods;
- we introduce a semantics for static analysis that preserves correctness while not imposing non-interference;
- and we identify areas of concern needing additional work.

2. PREVIOUS WORK

Although the limitations of JML's requirement that methods used in specifications be weakly pure are well known [9], only recently has significant effort been applied to the theoretical foundations of opurity. The theoretical work to date consists of a family of papers [2, 4, 14] with two threads of work, drawing on background work in simulation, information security, encapsulation, and representation independence. We will summarize that work here and draw on it heavily. For formal details and proofs, the reader is encouraged to consult the cited papers.

The theory concentrates on determining when side-effects of runtime assertion checking do not affect the execution and correctness of a program. To do so, we define a relationship $\asymp$ (read as *coupled*) among program states (Naumann's D-simulation, Barnett *et al.*'s C-simulation), such that states for which we expect the same behavior are related. For example, since weakly pure operations allocate new objects and change locations within such newly allocated objects, a coupling relation for weak purity would be such that if $h \asymp k$, then for all locations reachable from the domain of h, corresponding locations are in k with the same values, allowing only newly allocated objects in k and their fields to differ.

To define a suitable coupling relation $\asymp^S$ that ignores the side-effects of opure methods on secret state S, one could imagine relating any two states h and k that differ only in their values for the fields S (and are thus equivalent in their open state). But this turns

out to be too loose a condition. The secret portions of program state cannot be allowed to be arbitrarily different in general (unless that secret state is not used at all). Usually the secret state needs to be consistent in some manner with the open state. Technically, this condition is imposed by requiring that $\asymp^S$ is an observational congruence that is preserved by all statements and methods. Being an observational congruence for statements means that whenever $h \asymp^S h'$, each well-formed statement C preserves $\asymp^S$ (in the sense that executing C in h produces state k, then executing C in h' produces state k' and $k \asymp^S k'$). This preservation is enforced differently, depending on access to S:

- Each well-formed statement that does not directly access S must preserve $\asymp^S$. This condition prohibits $\asymp^S$ from exposing information about the structure of the secret part of the heap, S, and requires it to be an equivalence relation on open parts of the heap. This condition implies that, whenever $h \asymp^S k$, each expression that does not itself mention S returns the same value when run in both h and k, and that each method that does not access S also preserves $\asymp^S$.
- Each method that directly accesses S must be shown to preserve $\asymp^S$, and hence these methods cannot expose information about S to their callers. Since invariants can be formulated as boolean-valued methods, all invariants that relate the secret state to the open state must also be preserved by $\asymp^S$.

The prior work generally describes the $\asymp^S$ relation as parameterized by a class; however we define $\asymp^S$ with respect to a set of fields, S, that constitute the secret state.

Methods are related by $\approx^S$ if executing them on related ($\asymp^S$) states produces related states. Note that a method is not necessarily $\approx^S$ to itself, if it makes use of the differences in the secret state. Naumann shows that for weakly pure assertions, and given that there are no language constructs that expose the structure of the heap, replacing **`assert`** Q by **`skip`** in any context does indeed preserve equivalence of states, and thus weak purity is acceptable in specifications.

The theory continues with a definition of opure expressions. An expression E *is observationally pure with respect to* S if there is a coupling relation $\asymp^S$ such that execution of E preserves $\asymp^S$. That is, E only causes changes within S and those changes are consistent with the invariants that are part of the definition of $\asymp^S$.

To summarize this discussion of the prior work so far, a method m with side-effects on a portion S of the program state may be used in an assertion without jeopardizing correctness under the following circumstances:

(a) for each pre-state h (which includes m's arguments), the execution of m on h produces a post-state k such that $h \asymp^S k$, and
(b) every method of the program (including m) preserves $\asymp^S$.

The two threads of theoretical work diverge at the point of checking condition (a).

Equivalence to weak purity.

In Naumann's work [14], opurity is demonstrated by the following result:

> Suppose expressions (or procedures) M and N, acting respectively on states h and h', where $h \asymp^S h'$, produce states k and k', where $k \asymp^S k'$. Then if N terminates when M does and N is weakly pure, then M is observationally pure.

Thus to demonstrate opurity of a method, it is sufficient to find another method that is weakly pure and preserves coupling relationships as defined above.

Information flow.

In Barnett *et al.*'s work [2], opurity is demonstrated by an information flow analysis. In this case, one demonstrates that the result of executing a method is independent of any secret information to which the method may have access. An information flow analysis on the body of the method tracks which fields hold secret information, how that secret information is propagated, and how it affects control flow, in order to assure that the result of the method is not influenced by secret information.

That alone is too strict. In our cache example of Fig. 1, we definitely do want to be able to return the content of the secret cache when appropriate, as `getValue()` does. Hence Barnett *et al.* allow a method to return secret information if it can be demonstrated that the secret information is equivalent to information that is open.

3. OBSERVATIONS ON THE THEORY

3.1 Adjustments

In our view, the details of the theory described in the previous section could be profitably adjusted in two areas.

A minor issue is that the definitions given for observational purity in the related work do not explicitly require the returned result to be independent of secret state. This is an omission in Naumann's paper [14](Defs. 4.2, 5.3). In Barnett *et al.*'s work [2], opurity requires an accompanying simulation; it is not possible to define such a simulation if the purportedly opure method returned a result that varied with secret state. However, listing the requirement explicitly in the definition makes for easier static checking than if it is simply implicit in the required simulation.

Second, both papers do not consider executions in which some assertions are false, as they rely on a semantics of `assert` in which the assert statement does not terminate if the assertion expression is false. Barnett *et al.* [2] explicitly state that they only consider terminating computations; thus, only computations in which all assertions are true are considered. This vacuously precludes the possibility that an assertion might become false because of side effects of runtime checking. However, that is an important possibility that should be ruled out. Furthermore, in JML's runtime assertion checker, a false assertion does not terminate the program, but throws an exception. While technically JML's semantics also does not specify anything after such an exception (because the exception is a subtype of `java.lang.Error`), in practice one might continue to execute after catching the exception, and thus the program might check further assertions after it occurred.

3.2 Static Checking

The theoretical work described above focused on runtime checking. It derived conditions under which any side effects of executing assertions would have no effect on the open program state (up to an equivalence relation). However, this is a stronger condition than is needed for static checking. In static checking we need only know that using an opure method call in a specification does not make the meaning of what it specifies vary, depending on secret state. It is possible that executing an assertion would change the program state significantly and observably and yet the meanings of all specifications would remain unaffected and the program would always run correctly. That is, the stronger condition established by the simulation arguments in the prior work is sufficient to prove that correctness is maintained, but may not be necessary for correct static verification.

There has already been work on the semantics of using strongly and weakly pure methods in specifications [1, 6]. It is straightforward to replace the invocation of a program method m with a call

to an uninterpreted logical function f (whose arguments may include the program heap and the receiver object). The properties of f are given precisely by the specifications of m: f is well-defined when m's preconditions are satisfied and m terminates normally; the result of f is constrained only by m's normal postcondition. Thus an axiom for f can be created and used in verification against a specification that calls m. In the following, we call this semantics the *weakly pure semantics*.

What semantics should be used for calls to an opure method m'? A simple, intuitive semantics is to ignore the side effects of m' and produce the corresponding uninterpreted function f' and axioms exactly as in the weakly pure semantics. This is equivalent to replacing a call to m' by a call to a weakly pure method that has the same specifications as m'.

However, the weakly pure semantics does not necessarily match the behavior of runtime checking. To precisely model the complete execution of runtime checks, all specifications become assumptions and assertions, and methods in specifications are treated precisely as methods in program code are treated: locations that might have been modified by a method call are havocked - treated as undetermined except for the constraints of invariants or postconditions.

Recall that open methods may not reference secret state except through opure methods, and the results of these methods may not depend on secret state. Furthermore no presumptions are made about the secret portion of the pre-state other than that invariants are satisfied. Thus nothing in the execution of an open method can depend on the status of the secret state. It follows that prohibiting opure methods from directly referring to secret state in their pre- and postconditions guarantees that the weakly pure semantics can be soundly used for opure method calls in open methods.

The prior theory prohibits opure methods from being used in specifications in contexts that manipulate secret state, since such methods may themselves modify secret state, so non-interference cannot be assured. We argue below that soundness of static verification is preserved, even though non-interference is not, when query methods are used in invariants and method specifications of opure methods, so long as the method specifications do not also refer to secret state. In that case the same arguments as above hold, namely that a weakly pure semantics for opure methods is equivalent to a runtime semantics, for opure and secret methods.

Thus we conclude that, provided some restrictions are followed, in static verification one can safely model opure methods using the weakly pure semantics. The restrictions are that secret information cannot be accessed in opure method specifications and opure methods may not be used in assertions in the bodies of opure and secret methods that refer to the same secret state.

3.3 Theoretical extensions

The previous work has laid an excellent theoretical foundation. However there are some practical issues that require adapting and extending the results summarized above. Some additional points, unaddressed in this paper, are discussed in Sec. 6.

• Both of the main related works [2, 14] primarily use the class as the encapsulation unit. That is too coarse for practical use. The secret information is often restricted to one or just a few methods in a class and is not directly used by other methods. Hence, as both papers anticipate, we will define a smaller encapsulation unit.

• All of the previous work discusses the situation with secret state declared in just one class. In practice, a program will declare many pieces of secret state, associated with many different objects, and direct access to these pieces of secret state may occur in overlapping regions of the program. Note particularly that we define pieces of secret state as potentially associated with individual objects, rather than with a static declaration of a class or set of fields; in this way, operations on an object do not necessarily affect the secret state of other objects. We must ensure that the theory (and our language design) still applies in such situations.

• Naumann's work [14] prohibits all methods from exposing the values of secret state in a class, as this is required for proofs of observational purity via simulation outside that class. This is an overly strong restriction that we relax. For example, a class may define some helper methods that expose and manipulate secret state and that are intended to be used only within the implementation of opure methods.

4. APPLICATION TO JML

We apply the theoretical results above by making a number of modifications and translations within the context of JML.

4.1 Syntax

JML retains the **`pure`** modifier to mean weakly pure, as before. We add to the grammar of JML as follows.

- There are two new modifiers, **`secret`** and **`query`**, so the nonterminal *jml-modifier* [11] now has the additional options `secret` and `query`.
- In the package `org.jmlspecs.annotations` there are new annotation types: `Secret` and `Query`. Each may take a single argument, named `value` (so the key may be omitted) that is a String naming a secret datagroup; the default value of the parameter is an empty String. The name may be fully qualified or it may be unqualified. An unqualified name is, as usual, made into a fully qualified name by prepending the fully qualified name of the class containing the annotation.

As an example, the code in Figure 1 is annotated according to the proposed design.

4.2 Design and Semantics

This subsection describes the basic components of our design for JML and basic semantic checks.

The intent of our design is to partition the set of fields into two groups, open and secret, and to partition the set of methods into three groups: an open group that does not directly manipulate secret state, a secret group that can abstract manipulation of and access to secret state, and a query group that can also access and manipulate secret state, but in a way hidden from calling methods. Secret fields and methods constitute, use, or expose secret state. Query methods are intended to be opure. All methods and fields that are not annotated with **`query`** or **`secret`** are open.

4.2.1 Modifiers Themselves

The **`secret`** modifier is equivalent to a `Secret` annotation with no argument; these may be applied only to declarations of the following:

- a field, including datagroups, ghost and model fields, and
- a method (but not a constructor).

The `Secret` annotation with an argument can only be applied to method declarations.

The **`query`** modifier is equivalent to a `Query` annotation with no argument. The **`query`** modifier or annotation may be applied only to declarations of a class, interface, or a method; neither may be applied to a constructor or a field declaration.

4.2.2 Secret Fields and Datagroups

Groups of secret fields are used to define encapsulation boundaries for opurity. This is a difference from related work on opurity, which uses classes. Since JML already uses subtype-extensible "datagroups" [12] to group fields for purposes of specifying frame axioms (i.e., what fields a method may modify), we reuse this concept to group fields for defining encapsulation boundaries. Nontrivial datagroups are typically declared as model (specification only) fields. JML also has a type `JMLDataGroup` that can be used to declare such model fields. Note that instance (non-static) datagroups are associated with individual objects, not with the class as a whole.

A secret datagroup is declared using the `Secret` annotation or modifier on the datagroup's declaration. A field f is a *secret field* if it is declared using the `Secret` annotation or modifier.

A secret datagroup may contain only secret fields or other secret datagroups. A field must be in a secret datagroup and may not be in an open datagroup.

For a given query method we require there to be a secret datagroup G that contains the (secret) fields that constitute the secret state and that the query method might modify. As we will see, methods must be declared as either secret or query for datagroup G if they directly access (i.e., read or write) members of G.

Secret fields may not be used in the program or specifications of open methods; secret fields may not be used in the method specifications of query methods.

4.2.3 Pure and Query Types

A type, i.e., a class or interface, may be declared using the keyword **`pure`** in JML. Such a type is called a *pure type*. In a pure type all methods not declared as query methods are implicitly declared to be pure [11].

We extend this convention to the `Query` annotation: in a *query type*, all methods not declared as pure are implicitly declared as query methods with the same query keyword or annotation.

(We do not allow secret types.)

4.2.4 Pure Methods

Weakly pure methods are still declared using **`pure`**. However, the current rules for (weakly) pure methods are changed slightly.

A method is a *pure method* iff it either: (a) has a **`pure`** modifier or annotation, (b) overrides or implements a pure method, or (c) is declared in a **`pure`** type and neither overrides a query method nor has a **`query`** modifier or annotation.

4.2.5 Secret Methods

A method declared with the secret annotation (or modifier) indicates that the method can directly access and modify some secret datagroup in a way that need not be opure.

A method m is *a secret method for datagroup G* if one of the following holds.

- Method m is declared with the annotation `@Secret("`G`")`.
- Method m is declared with the **`secret`** keyword or with the annotation `@Secret` (with no arguments), and all the methods that m overrides are secret methods for datagroup G.

Note that if a secret method does not override a secret method, it must use an annotation that names a datagroup. This datagroup must be visible whenever it is used. Thus, if a method is declared secret for G in multiple interfaces and classes, the datagroup G must be visible at all the declaration sites.

A secret method may not override or be overridden by a non-secret method.

A method may not be both a query method and a secret method for the same datagroup.

Secret methods may not be used in the program or specifications of open methods; secret methods may not be used in the method specifications of query methods.

Finally, it must be shown that any changes by a secret method to open state must be independent of the secret state. For this we require that any such changes be specified using open computations.

4.2.6 Query Methods

A method declared with the query modifier or annotation indicates that the method must be observationally pure with respect to some secret datagroup.

A method m is *a query method for datagroup G* if one of the following holds.

- Method m is declared with the annotation `@Query("`G`")` (or is declared in a class with this annotation and is not declared with the modifier **`pure`** nor overrides a **`pure`** method).
- m overrides some method, and each method m' that m overrides is a query method for datagroup G. m optionally but preferably has a query annotation or modifier.
- Neither of the above applies, method m is declared with the **`query`** keyword or with the annotation `@Query` (with no arguments), and G has the same name as m. If there is no declaration of a datagroup with the same name as the method m in scope, then m's name is implicitly declared to be a secret datagroup in the same class as m, with the same Java visibility as the method, with the **`static`** modifier iff m is static, and with type `JMLDataGroup`.

If an implicit declaration of a datagroup with the same name as a query method would be illegal, then the query annotation must explicitly give the name of the associated datagroup.

A method may not be declared with both query and pure modifiers; furthermore, a query method may not override a pure method. However, a pure method may override a query method.

A query method must have a specification, and that specification must contain at least one normal-behavior specification case.

The default assignable clause for a specification case of a query method for datagroup G is `assignable` G. If an assignable clause is given for a query method, it may contain only secret fields or datagroups. Furthermore, a query method m may only directly modify (at most) the pre-state fields in the datagroup with which it is associated (it is effectively an open method for other secret datagroups and may not directly read or write the secret fields of those other secret datagroups).

Finally, the return value of a query method for a datagroup G must be shown to be independent of G. This can be established, per [2], by proving that the returned result is equal to the result of an open computation. This will be trivial if the query method's postcondition has a form such as **`ensures \result==`** . . . . In such cases the condition that establishes that the return value is independent of secret state is the same as the postcondition, since we require that the postcondition does not read any secret state. (There is no point to a **`void`** query method.)

4.3 Rules for Legal Use

In this subsection we describe how other parts of JML interact with secret and query fields and methods.

4.3.1 Static and Instance Datagroups

The discussion above extended the use of secret state to multiple, disjoint datagroups of secret state within one program. As long as the datagroups are disjoint we can treat a method as opure for one datagroup but open for others.

Most datagroups are sets of instance fields belonging to a given object. Then the datagroups for two different objects, even of the

same type, are disjoint. A method that is opure for one object is open for the other and can be used without restriction in the second object's specifications. The **`equals`** method, for example, can be declared query for the non-static **`Object.equals`** datagroup. The **`equals`** method is restricted from being used in the body of query methods of its own overriding methods *for the same object*, but **`equals`** can be called on other objects. Thus **`equals`** for a Collection can make use of **`equals`** for its elements, as long as the Collection object is not an element of itself.

In order to enforce this disjointness, static fields may not be elements of secret instance datagroups. Also, pending further experimentation, secret fields are forbidden to be members of two different secret datagroups where one is not a subset of the other.

4.3.2 Use in Type-Level Specification Clauses

Invariants and (history) constraints may directly read secret fields and call pure secret methods, but only those declared in the same type or a supertype. No other type specifications (including initially, represents, monitors-for, readable-if, and writable-if clauses and axioms) may directly read secret fields or call secret methods.

A query method may be called in invariants and constraints of type specifications of any type. A query method for a datagroup G declared in a type T may also be called from within other type specification clauses, but such calls are prohibited from subtypes of T (including T itself).

By the theory above, a query or secret method m for datagroup G is not allowed to call a query method p for G (including m itself), on the same object, in its own type or method specification. We relax that rule to allow query methods in type and method specifications according to the following argument.

In statically verifying a method m, the invariants and method specifications are assumed at the beginning of the body and asserted at its end(s). The final assertion cannot affect the course of the execution within m, and the theory has established that it is immaterial to open methods calling m. Query methods called prior to the body of m may alter secret state, but must maintain the invariants that apply to the secret state. As long as there are no direct references to secret fields in the type or method specifications, there can be no interference in any runtime checking of those specifications. Thus, at the beginning of the method's body, the invariants will still hold and coupling will be preserved. Static verification will only assume that the invariants hold and will not assume any more specific information about the secret state. By assuming a weakly pure semantics for query methods combined with no knowledge of secret state, we verify a conservative approximation to any runtime execution that agrees with the specifications.

In runtime checking, the invariant is asserted as part of checking the preconditions. This may well alter the secret state in a way that is visible within the body of m; for example, only a part of the control flow may ever be executed. However, presuming that the static verification shows that the method is correct, executing the invariant at runtime will not affect the truth of any assertions executed in the body of the method.

Thus we conclude that using query methods in type and method specifications is permissible. A particular invariant may not both call query methods and also call secret methods or use secret fields; query method specifications may not refer to secret methods or fields at all; secret method specifications may not mix query methods and secret fields or methods.

An alternate reasoning for this conclusion provides a different perspective. Consider two sorts of statement sequences: (A) statements that call opure methods but do not access secret state, and (B) statements that access secret state but do not call opure methods. The prior theory demonstrated that open methods, which consist of type (A) sequences, preserve correctness; it also demonstrated that opure methods, which consist of type (B) sequences, preserve correctness as well (presuming in each case that the methods maintain instance invariants). Now an opure method with specifications consists, during runtime checking, of invariant checks, precondition checks, the method body, postcondition checks and invariant checks. This is equivalent to a sequence of method calls; it will preserve correctness if each step is either type (A) or (B), as noted in the conclusion above.

4.3.3 Use in Method Specifications

Secret fields and methods for a datagroup G may be accessed or called in method specifications only by secret methods for datagroup G. Method specifications of non-secret methods must not read secret fields or call secret methods.

As concluded in the previous section, query methods for a datagroup G may be used in any method specification that does not also access secret methods or fields for G.

However, assignable clauses in any method specification may mention secret datagroups, if the method being specified calls query methods in its program or specifications (but see the discussion in section 5.2).

In the context in which a query method is used within a specification (e.g., taking into account any short-circuit guards), the precondition of at least one normal-behavior specification case must be satisfied. This establishes that the execution of the method is well-defined, just as well-definedness requires that the receiver of a field selection operation is non-null [5]. A query method's semantics are generated only from its normal-behavior specification cases. (Pure methods should obey a corresponding rule.)

4.3.4 Use in Method Bodies

Secret methods and fields may not be used in the Java programs or JML assertions in the bodies of open methods. Query methods may not be used in assertions in the bodies of query or secret methods for the same datagroup, but may be used in secret and query methods for other datagroups. Query methods may be used in the Java code of method bodies. This is a major benefit of basing encapsulation on datagroups, since one can use one query method in the body specifications of another, as long as they are associated with (different instances of) different datagroups.

4.3.5 Use in Constructors and their Specifications

Constructors are open and may not be declared to be query or secret. Specifications of constructors may not directly refer to secret fields or call secret methods. However, constructor specifications do implicitly include any invariants that mention secret fields.

A constructor's body may read, write, or call any open field or method, any secret field declared in its class, and any query or secret method for a datagroup that is declared within its class. Any secret fields hold default values when a constructor begins executing, so there is no secret state information to leak.

A constructor may not call secret methods or access secret fields of its superclasses, as that secret state is already initialized.

To avoid interference, query methods may not be used in assertions within the body of a constructor.

Specifications of constructors may not directly refer to secret fields or call secret methods. However, constructor specifications do implicitly include any invariants that mention secret fields.

5. DISCUSSION

We expect our design to provide a means of using simple opurity patterns while providing a platform for further experimentation. A key test of this proposed design is usability: assessments of its practicality on larger code bases than test examples are underway. In particular, it serves well for specifying library classes such as the JDK whose implementation is unknown but whose user-supplied overriding methods may be opure.

A key aspect of this design is that it accommodates disjoint sets of secret state and that those sets may be associated with individual objects. Although the previous theory applies in large part unchanged, it presumed a syntactically defined encapsulation boundary. This allows us to call opure methods on one object in the execution of the method on another, at the cost of proving that the objects are different instances.

Another interesting aspect of the design is that secret fields and methods may be public. This is intentional, as it allows existing code to be annotated in a way that preserves observational purity. If such annotations can be given in a way that follows our design, then it will be safe. However, we do recommend that secret fields and methods not be public.

One does need to plan ahead for opurity. If a method is ever going to be overridden and implemented using some secret state, it must be declared a query method from the start. It cannot be pure in a superclass and query in a derived class. This means that nearly every method that might be used in a specification should be declared query rather than pure. That is why the implicitly declared secret datagroup for a method is part of the design—so that the only specification needed in the simplest case is to declare a method as a query method.

5.1 Annotations

The secret annotations enable a simple level of encapsulation. This is sufficient for the more common examples of opurity. Further use will show whether this is adequate for large-scale software systems.

It may be that experience will show the need to be able to use **`secret`** as a type modifier and, for example, be able to declare local variables and formal parameters as secret, in support of detailed information flow analysis. For now we rely on proofs that any assignments to non-secret fields consist of open information. Since this is only needed for the return result of query methods and for secret methods that might assign to open state, we expect the need for a full information flow analysis to be rare.

5.2 Frame Conditions

So far, nothing we have established about opurity has changed the rules regarding frame conditions: each method must declare those fields that it might modify, either directly or indirectly through methods it calls. However, consider the following. Since methods that override `Object.equals` may and do modify secret state, `Object.equals` must be declared a query method and specified that it might modify the method's secret datagroup. A library method `HashSet.contains` presumably uses `equals`, although its implementation may not be known. If it does, it would need to declare that it might modify `equals`'s datagroup. These frame conditions will propagate everywhere. Requiring them for secret state that users do not know or care about would be a decided inconvenience.

So may the frame conditions regarding secret state be omitted? If we do so, then we must assume that any method may indirectly modify any secret state in the program. That is, every non-pure method in the program implicitly has a frame condition that allows modification of any secret state. The modifications still preserve invariants, however. Thus after any method call in a program, we can assume that any secret state still obeys its invariants, but is otherwise undefined. This is acceptable for open methods that do not access secret state anyway. However, it would be a complication for query and secret methods that are manipulating secret state. Query methods associated with the same secret state were already prohibited in assertions in a method body. But now any method at all, including for example `Object.equals`, may affect the secret state at hand. This analysis is independent of whether the methods are called in the program or in specifications.

It is sound to consider that any method may modify any secret state. However, it may complicate writing and verifying methods that manipulate secret state, since all secret fields are then essentially volatile. Investigation is underway to determine whether this approach is usable. The complications may be particularly complex if secret state is nested and layered. For example, it may be desirable to allow a particular piece of secret state to be declared as either (a) part of the global secret state and thus allowed to be implicitly part of every non-pure frame condition or (b) not part of the global secret state and required to be explicitly listed in frame conditions as needed.

6. FUTURE WORK

There remain a number of open theoretical issues for future work. Chief among them are the usability of implicit assignable clauses and how to handle nested or shared secret state. Furthermore, as always when theoretical work is extended for practical application, there is the task of formalizing and proving the extensions and establishing soundness of the design as actually used; this is particularly the case for our informally argued conclusions about the semantics of opure methods in static analysis and the use of opure methods in invariants and method specifications.

The discussion so far has treated information as strictly either secret or open. In practice, however, it is the observation made of the secret information that is of consequence. For example, the hashcode method produces an `int` whose value depends on the state of the heap. Thus hashcodes will change if weakly pure assertions are executed. However, all that we *use* of a hashcode is its invariant property: if two objects are equal then their hashcodes are equal. Indeed it is this reasoning that is behind allowing location remapping when comparing program states. Absolute location is unimportant; all that matters is location equality. We need a semantics of secret information that allows for equivalence of program state in terms of predicates over state rather than equivalence of open state.

From an external perspective all that is observed of a program is its input and output. In many cases both are textual, including anything displayed in a GUI. From this perspective, everything within a program is unobserved state. Within any program with any degree of abstraction there will be nested layers of hidden information. The work described above needs to be extended to situations in which various groupings of secret information occur in nested layers. We leave for future work the handling of nested secret state, but we expect it to have close relationships with other encapsulation disciplines, such as the pack/unpack facility in Spec# or the universe type system.

7. CONCLUSIONS

We have adapted and applied the previous theoretical work on observational purity to a proposed practical design within JML. In the process we have defined the encapsulation boundary to include precisely the secret state and extended the theory to accommodate multiple groups of secret state, secret helper methods, invariants,

method specifications and implicit frame conditions. We have also introduced a semantics for static analysis of opure methods that relaxes non-interference while maintaining correctness.

The design above is implemented in OpenJML, an experimental version of JML at the level of Java 1.6, built on the OpenJDK code base.

The lack of observational purity in specification languages has been one roadblock to widespread use of source-level specifications on substantial code bases. Providing a design and implementation in JML will allow experimentation with such code bases.

Acknowledgments

The work of Leavens has been supported in part by grants from the US National Science Foundation numbered CCF-0428078, CCF-0429567, and CNS 08-08913. Thanks to David A. Naumann for private communications clarifying some aspects of his paper on opurity [14].

8. REFERENCES

[1] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, June 2006.

[2] Michael Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. Allowing state changes in specifications. In Günter Müller, editor, *ETRICS*, volume 3995 of *Lecture Notes in Computer Science*, pages 321–336. Springer, 2006.

[3] 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, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[4] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specification. Obtained from the following URL: `http://guinness.cs.stevens-tech.edu/~naumann/publications/purityJoT.pdf`, January 2005.

[5] Patrice Chalin. A sound assertion semantics for the dependable systems evoluation verifying compiler. In *International Conference on Software Engineering (ICSE)*, pages 23–33. IEEE, May 2007.

[6] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[7] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[8] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.

[9] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Lieden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 262–284. Springer-Verlag, Berlin, 2003.

[10] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2008.

[12] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

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

[14] David A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 376(3):205–224, 2007.

Component-Based Design in Tako: A Case Study

Arun Sudhir
Virginia Tech
Falls Church, VA
aruns@vt.edu

Gregory Kulczycki
Virginia Tech
Falls Church, VA
gregwk@vt.edu

Jyotindra Vasudeo
Virginia Tech
Falls Church, VA
vasudeo@vt.edu

ABSTRACT
Tako is an object-oriented language similar in many respects to Java, but is designed to support alias avoidance and thereby simplify both formal and informal reasoning. Aliasing in Java occurs mainly due to reference assignment, which Tako replaces with alternative data assignment mechanisms such as copying, swapping, and initializing transfer. Though the changes are syntactically minor, their effect on component design and design patterns is not. This paper examines a non-trivial program designed and implemented in Tako, and discusses how and where the design differs from a typical Java program. We look at how the design impacts specification and reasoning. We found that while many design decisions were unaffected by the emphasis on alias avoidance, there were certain design issues that Java programmers would need to adjust to. A key component in the example program is a tree data structure that would likely be implemented using a composite pattern in Java.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features.

Keywords
Alias avoidance, design patterns

1. INTRODUCTION
Tako is an object-oriented language with Java-like syntax that supports alias avoidance, thereby simplifying both formal and informal reasoning [1]. Though most alias-avoidance languages, including Tako, do support limited aliasing, they differ from alias-control languages in that aliasing is the exception rather than the rule. Tako's main use to date has been as an instructional tool to help teach students how to reason formally about their code. The students are taught how to read and write specifications, and how to use those specifications to trace through code based on specific input. They also learn how to construct symbolic reasoning tables – generalized, user-friendly, tracing tables that can be used to generate the verification conditions needed for correctness proofs. Tako simplifies writing specifications and reasoning about code because programmers do not need to keep track of the indirection that pervades traditional object-oriented languages.

Many of the alias-avoidance techniques found in Tako have their origins in the Resolve language [2, 3]. Resolve is an integrated programming and specification language intended to support full, heavyweight program verification. Central to the approach of both Resolve and Tako to facilitate alias avoidance is the use of alternative data assignment operators such as swapping. Some researchers have raised concerns about whether the paradigm associated with this approach "can mesh well with mainstream object-oriented programming techniques" [4, 5].

This paper examines a non-trivial program designed and implemented in Tako, and discusses how and where the design differs from a typical Java program. We found that while many design decisions were unaffected by the swapping paradigm and the emphasis on alias avoidance, there were certain design issues that Java programmers would need to adjust to.

Section 2 gives a brief overview of the Tako language, emphasizing how it differs from Java. Section 3 describes the architecture of the program we designed – a simple text-based adventure game. Section 4 describes and partially specifies a key data structure used in the program, an indexed tree, which has the features of both a tree and a map. Section 5 describes how the indexed tree is used in the program and demonstrates how to trace through a portion of code based on the indexed tree specification. Section 6 raises other design issues that distinguish Tako from Java. Section 7 provides some concluding thoughts on the subject.

2. OVERVIEW OF TAKO
The main difference between Tako and Java is that Tako includes alias avoidance features. This allows programmers to view variables directly as objects rather than as references to objects. The following subsections give a few important differences.

2.1 No primitive types
In Java, there are two kinds of types: primitive types and reference types. Primitive types are built-in to the language and their variables denote values. Some reference types are built-in to the language, but most are user-defined. Variables of a reference type denote references to objects. In Tako, all types are value types. Some are built-in and others are not, but variables in Tako always represent objects, no matter what type they are from. In addition, no two variables ever represent the same object.

In Tako, as in Java, some types that are built-in to the language have special syntax. These include Booleans, Integers, Strings, and Arrays. In general, if a type has special syntax in Java, its corresponding type in Tako will probably have it also.

Sometimes we talk about replicable types in Tako. A type is replicable if it has a *replica* operation. Some common types like Booleans, Integers, and Strings, already include a *replica* operation. Programmers can make any type replicable by simply adding a *replica* operation themselves.

2.2 Initial values
In Java, the compiler will report an error if you try to use a variable before you have initialized it. In Tako, all variables get

initial values when they are declared. Tako uses the default constructor for this purpose. As in Java, Tako programmers are encouraged to provide default constructors for all objects. If no default constructor exists for a type, a newly declared variable of that type will get a null value. Since null values are not consistent with viewing variables directly as objects, omitting a default constructor is discouraged.

2.3 Alternative data assignment

Java's assignment operator introduces aliasing because it copies references. Tako is designed to avoid aliasing, so it requires alternative mechanisms to assign objects to variables. Here is a brief overview of the alternatives.

2.3.1 Swapping

Swapping is the primary means of data assignment in Tako. When two variables are swapped, the variables simply exchange objects. Swapping does not introduce aliasing because if the variables denote distinct objects before the operation, they still denote distinct objects after the operation. Swapping is also a constant time operation, because the compiler implements it by swapping memory locations. However, swapping is a symmetric operation, so both variables need to have the same type before they can be swapped.

2.3.2 Initializing transfer

The initializing transfer operation in Tako "<–" transfers an object from one variable to another and gives the first variable an initial value. The transfer operation is fairly efficient, but if the variable receiving the object already had a different one, its original object will become garbage and will have to be deallocated eventually.

2.3.3 Function assignment

Another way of getting an object into a variable is by assigning the result of an function to the variable. The function assignment operator in Tako ":=" always expects a variable on its left-hand side and an expression on its right. If the compiler sees anything other than a variable on the left-hand side, it will complain. If it sees a variable rather than an expression on the right-hand side, as in "max := n", the compiler tries to replicate the variable, as in "max := n.replica()". If no replica operation is found, it reports an error.

2.4 In-out parameter passing

By default, parameter passing in Tako is in-out. In other words, argument values are transferred to the formal parameters, the method is executed, and formal parameter values are transferred back to the arguments.

In-out parameter passing allows Tako programmers to keep functions and procedures distinct. A function has a return type (non-void) and a procedure does not. By convention, functions should not have side-effects. A function has side-effects if it changes the value of a variable. An example of a side-effecting function is a pop method in a Java stack, as in the assignment x = s.pop(). It is a function because it returns a value, and it has a side-effect because it changes the current stack object. If a Tako programmer wants an operation to change the state of the program, they should write it as a procedure rather than a function, so that it would be called as "s.pop(x)".

2.5 Result variable

In Tako functions (non-void methods), the result of the function is returned through a special result variable. This guarantees that the object returned is unique and is not an alias to any existing object. The result variable has the same type as the return type of the function. The compiler treats the result variable as if its declaration is the first statement of the method. So a getter method for a private attribute length would be written as "**public** Integer getLength( ) { **result** := length; }" and interpreted by the compiler as "**public** Integer getLength( ) { Integer **result**; **result** := length; }". The result variable is initialized when it is declared, so even a function with no statements would return an initial value.

2.6 Pointer component

Despite the fact that the design of Tako is focused on avoiding general aliasing, we understand that there are circumstances when programmers will need pointers and references to efficiently implement certain classes. For this purpose, Tako has a pointer component that is specifically designed to aid in the implementation of linked data structures such as lists and trees.

3. ADVENTURE GAME ARCHITECTURE

To experience first hand the paradigm shifts involved in programming a non-trivial application in Tako, we undertook the development of a text-based adventure game. The game was initially developed in Java, but with the intent that it would eventually be ported to Tako. Figure 1 shows the general architecture of the application in the form of a UML class diagram. It is loosely based on traditional text-based adventure game development systems such as Inform [6] and TADS [7].

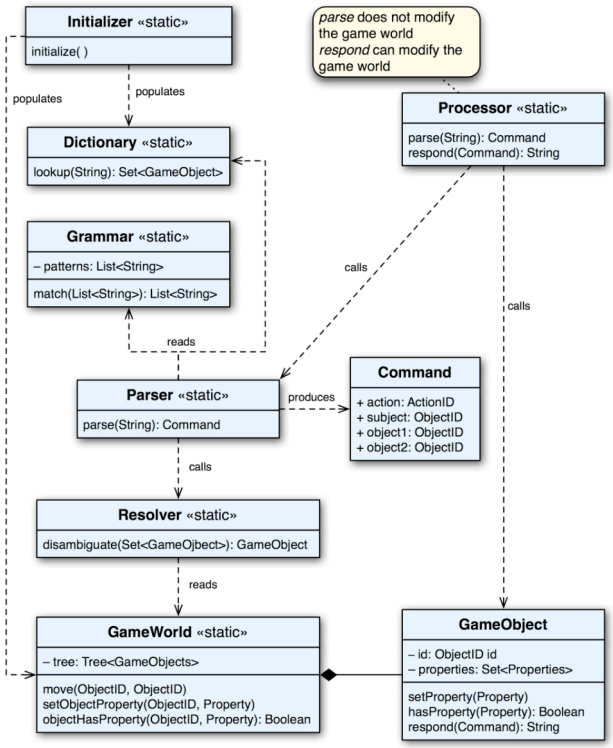


Figure 1. Adventure game architecture

The game accepts text inputs from the player, which are usually simple imperative sentences, such as "take the chess piece" or "put the red queen on the chess board". A Parser component parses the input based on a supplied grammar and dictionary of game objects. A Resolver component tries to determine what game object is intended when the player enters ambiguous text. The GameWorld component tracks the state of the game. It is a cross between a tree-like data structure and a database that stores all the game objects. When the player inputs a command, the application updates the GameWorld accordingly and generates a text response.

The program contains about 50 classes and consists of over 4,000 lines of code. It required approximately 85 man hours to code the game in Tako based on the Java version of the game. Table 1 gives process metrics for the conversion from Java to Tako. The time spent on the conversion is shown during and after the translation process. During the translation process, most of the time went into translating statements that used reference copying into statements that used swapping. Part of this process involved direct substitution of the reference copy operator with the swap operator. Part of it also involved swapping objects from containers. In both cases, there was the possibility that objects had to be swapped back, as illustrated in section 5. A fair amount of time was also spent in converting methods with return values to their equivalent in Tako. If the methods had side-effects, then they where changed to procedures (methods without return values) in Tako, and the return value was passed out through a parameter. If the methods did not have side-effects, then the methods where changed to appropriate functions (methods with return values) in Tako. In Tako functions, the distinguished **result** variable is used to store the return value. Some time was also spent for converting Java enumeration types to static integer variables. Enum types are supported in Java 1.5 but not in Tako. The remaining time was spent in copying and pasting code from one language to another. This was possible due to the similarities in Java and Tako syntax.

Table 1. Process metrics for conversion from Java to Tako

Description	Hours
Time spent during translation	
Conversion of Enum types to static integer variables	2
Converting side-effecting functions	10
Converting non-side-effecting functions	5
Translating code with aliasing to swapping methodology	15
Simple translations (copy and paste)	5
Time spent after translation	
Debugging errors due to erroneous translation	30
Debugging errors already present in Java version	18

Debugging code after the translation took up the majority of time. The debugging process metrics were divided into two parts: time spent in debugging errors that occurred due to erroneous translation, and time spent in errors that where present in the original version of the Java code. Nearly 30 hours were spent in debugging the translation errors. We had expected this part of the process would take the most time since this was our first attempt at such a translation. The other 18 hours spent in debugging could have been avoided if the original Java version had been tested thoroughly.

4. INDEXED TREE COMPONENT

The two most sophisticated components in the adventure game are the Parser and the GameWorld. The Parser takes an imperative sentence typed by the player and converts it to a four-part command. The parser as implemented in Tako is not very different from the parser as implemented in Java. This is probably due to the fact that the Parser is designed to essentially encapsulate a single, though complex, method – parse. The Tako GameWorld component does have significant differences with the Java GameWorld component. Therefore, we spend this section and the next discussing it. The GameWorld component is based on a custom data-structure called an IndexedTree.

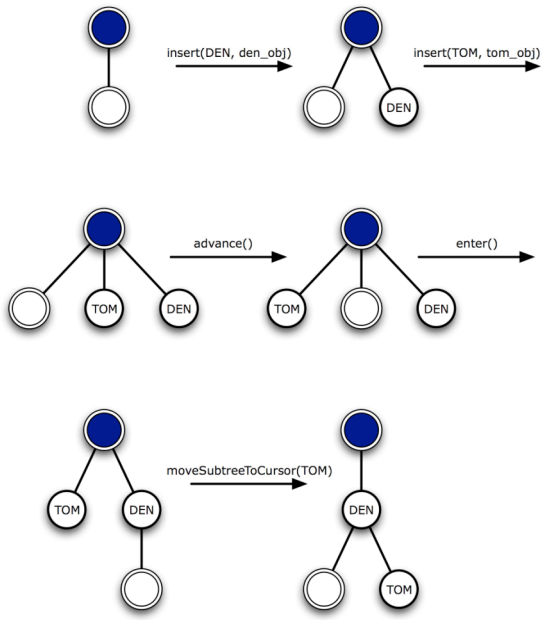


Figure 2. Tree methods

The elements of an indexed tree are organized as an ordered tree [8]. An ordered tree contains a root node, which is the ancestor of all the other nodes in the tree. Every node except for the root has a parent. Nodes with the same parent are siblings. In an ordered tree, the siblings are ordered. There is a first, or eldest, sibling; and there is a last, or youngest, sibling. The indexed tree data structure is traversable. That is, a tree has a conceptual location known as the cursor position. We conceptualize the cursor position as a distinguished node in the ordered tree. The tree component provides various cursor movement methods that can be used to easily change the cursor's location in the tree. Insertion and removal of nodes from a tree occurs to the right of the cursor. The component is called an *indexed* tree because all tree nodes are indexed, or labeled, with a unique identifier. This allows individual nodes to be accessed directly.

Figure 2 gives a graphical representation of various states of an indexed tree object and shows the method calls that cause the transitions from one state to another. The first tree in this figure represents an initialized tree. It is the tree that is created when the default constructor is called. It has two nodes – a root node and a cursor. The cursor is the child of the root. The call insert(DEN, den_obj) inserts a new node to the right of the cursor. The new node is the cursor's younger sibling. The new node DEN is associated with the object den_obj. A second call to insert with

label TOM and object tom_object inserts a new node to the right of the cursor, just before the DEN node. The call advance() advances the cursor past its next node, TOM, and enter() causes the cursor to enter the subtree induced by its next node, DEN. A somewhat more sophisticated method call, moveSubtreeToCursor(TOM), causes the subtree induced by TOM to be moved just to the right of the cursor. It requires that the TOM be in the tree and that the cursor is not a descendent of TOM.

4.1 Mathematical Model for Indexed Tree

Figure 3 gives the mathematical model for the indexed tree component. It contains four model variables that specify how an indexed tree object is modeled [9]. The first three variables are based on the mathematical model for ordered trees given in Cormen et al. [8], in which graphs are modeled as two sets – a vertex set and an edge set – and a tree is an acyclic, undirected, connected graph. The keyword model indicates that a variable is part of the mathematical model. The variable *nodes* denotes the vertex set, and *edges* denotes the edge set. The vertex pairs in the edge set are unordered, so edge (3, 1) is the same as edge (1, 3). The variable *order* denotes the order for children of the same parent from eldest to youngest. The order of the eldest child is 1, the second eldest child is 2, and so on. The final model variable, *contents*, maps nodes to objects.

```
public interface IndexedTree {
   model nodes:     set of Enum;
   model edges:     set of pair of Enum;
   model order:     function from Enum to Integer;
   model contents:  function from Enum to Object;
   defines ROOT, CSR: nodes;
      constraints /* no cycles */

      public IndexedTree( );
         ensures nodes = { ROOT, CSR } and
            edges = { (ROOT, CSR) } and
            order = { (ROOT, 1), (CSR, 1) } and
            contents = { (ROOT, null), (CSR, null) };
```

Figure 3. Model and constructor for indexed tree

The defines clause defines two distinguished variables that belong to the set *nodes*. Conceptually, ROOT is the root node, and CSR is the cursor. A class invariant (given by the constraints clause) asserts that no cycles exists in the undirected graph represented by the nodes and the edge set. The assertion is given here informally.

The constructor creates an indexed tree with ROOT and CSR as nodes, and an edge connecting ROOT to CSR. Both root and cursor have an order of 1 and map to null objects.

4.2 Cursor Movement Methods

A key feature of the tree data structure is the flexibility of cursor movement. Figure 3 specifies some selected cursor movement methods. For the other cursor movement methods, see [10].

The advance method advances the cursor to the next node on the same level. It requires that the cursor have a younger sibling to advance past. In the ensures clause, a variable with a hash, such as #nodes, refers to its original (or old) value, and a variable without a hash refers to its current (or new) value. The only change in the state is that the cursor and its immediate younger sibling, #next(CSR), get their orders swapped.

```
public void advance();
   requires hasYoungerSibling(CSR);
   ensures nodes = #nodes and edges = #edges and
      order(x) = (  #order(x) – 1   if x = #next(CSR);
                    #order(x) + 1   if x = CSR;
                    #order(x)       otherwise ) and
      contents = #contents;

public void enter();
   requires hasYoungerSibling(CSR);
   ensures nodes = #nodes and
      edges = #edges  minus { (#parent(CSR), CSR) }
                      union { (#next(CSR), CSR) } and
      order(x) = (  1                 if x = CSR;
                    #order(x) – 1     if #isYoungerSibling(x, CSR);
                    #order(x) + 1     if #isChild(x, #next(CSR));
                    #order(x)         otherwise ) and
      contents = #contents;

public void moveBefore(restores Enum key);
   requires key in nodes;
   ensures nodes = #nodes and
      edges = #edges  minus { (#parent(CSR), CSR) }
                      union { #parent(key), CSR) } and
      order(CSR) = (
         #order(#key) – 1  if #isYoungerSibling(key, CSR);
         #order(#key)      otherwise ) and
      order(key) = (
         #order(#key)      if #isYoungerSibling(key, CSR);
         #order(#key) + 1  otherwise ) and
      order(x ≠ CSR, key) = (
         #order(x) – 1     if #isYoungerSibling(x, CSR) and
                              not #isYoungerSibling(#key, x);
         #order(x) + 1     if #isYoungerSibling(x, #key) and
                              not #isYoungerSibling(CSR, x);
         #order(x)         otherwise ) and
      contents = #contents;
```

Figure 4. Cursor movement methods

The enter method makes the cursor the first child of its next node. It requires that the cursor have a younger sibling. The nodes and contents remain unchanged. The original edge involving the cursor is replaced by an edge from the cursor's original next sibling to the cursor. The original younger siblings of the cursor get their orders decremented. The cursor advances to the next level and becomes the eldest child of its new parent, so its order is 1, and the cursor's new younger siblings get their orders incremented.

The moveBefore method takes key as an argument. It requires that key be in the node set. It ensures that the cursor will be moved directly before key. That is, the cursor will become key's immediate older sibling. The node set does not change. The original edge to the cursor is replaced by an edge from key's parent to the cursor. If the cursor is the younger sibling of key, the cursor's order becomes one less than the original order of key and key's order stays the same. Otherwise, the cursor's order becomes the original order of key and key's order is incremented. For all other nodes, the order of the cursor's younger siblings are decremented, and the order of key's younger siblings are incremented. However, if a node is a younger sibling of both the cursor and key, its order remains unchanged. The contents map remains unchanged. The **restores** parameter mode for key indicates that the value of key remains unchanged, even though this is not explicitly stated in the ensures clause.

4.3 Insert and SwapValue Methods

Inserting elements into and removing elements from data structures affects how programs are designed in Java and Tako. In Java, updating an element inside a data structure means getting a handle to the element and updating the handle. This updates the element inside the data structure because the handle is an alias to it. In Tako, such aliasing is avoided. Therefore, the element must be removed from the data structure, updated, and put back into the data structure in the same place it was at originally.

```
public void insert(restores Enum key, clears Object val);
   requires key not_in nodes;
   ensures nodes = #nodes union { key } and
      edges = #edges union { (#parent(CSR), #key) } and
      order(x) = (
         #order(CSR) + 1    if x = #key;
         #order(x) + 1          if #isYoungerSibling(x, CSR);
         #order(x)              otherwise ) and
      contents = #contents union { (#key, #val) };

public void swapValue(updates Object val);
   requires hasYoungerSibling(CSR);
   ensures nodes = #nodes and
      edges = #edges and order = #order and
      contents = #contents override { (#next(cursor), #val) } and
      val = #contents(#next(cursor));
```

Figure 5. Insert and swap methods

The methods shown in Figure 5 modify the tree by inserting nodes and swapping values from it. We do not discuss how to remove nodes, but a description can be found in [10].

The insert method inserts a node into the tree as the immediate younger sibling of the cursor. key becomes the new node, and val is the contents of that node. The method requires that key is not already in the indexed tree. key is added to the node set, and an edge to key is added to the edge set. The order of key is one more than the order of the cursor, and the order of the nodes following key are incremented. The **clears** parameter mode for val indicates that val has an initial value after the call. Since the val object is inserted into the tree, the val parameter must hold a different object after the call. Were it to have a *restores* parameter mode, like key, it would force the implementer to perform a deep copy of val, which could be a potentially expensive operation. The key object is also inserted into the tree, but key is a small object (an Enum) so copying it is inexpensive.

The swapValue method swaps the contents of cursor's next node with val. It requires that the cursor have a younger sibling. It ensures that the node set, edge set and order map remain unchanged. The existing contents of the node gets the original object in val, and val gets the original contents of the node. The **updates** parameter mode indicates that the value of val is updated.

5. USING THE INDEXED TREE

5.1 The GameWorld and its GameObjects

The IndexedTree component is used in the implementation of the GameWorld component. The game world is an indexed tree whose nodes are identifiers that are mapped to game objects. A game object inherits from the GameObject class. The GameObject class includes two fields: one for a unique identifier, and another for a set of properties, as shown in Figure 6. Both ObjectID and Property are enumeration types.

```
public interface GameObject {
   model id:          ObjectID;
   model properties:  set of Property;

   public GameObject( )
      ensures id = VOID and properties = { };

   public void addProperty(restores Property p)
      ensures properties = #properties union { #p };

   /* other operations */
}
```

Figure 6. GameObject specification

An object identified by DEN of type Room might include the property LIGHT so that players can see objects in the room. An object identified by TOM of type Actor might include the property PERSON so that the player can talk to it, and the property MALE so that the game's printer knows what pronoun to use when referring to the object. An object identified by BOX might include the property BIN so that players can place other objects inside it. If it has the property OPEN a player may be able to see its contents.

```
public class GameWorld {
   model nodes:    set of ObjectID;
   model edges:    set of pair of ObjectID;
   model order:    function from ObjectID to Integer;
   model contents: function from ObjectID to GameObject;
   defines ROOT: nodes;
      constraints /* no cycles */

   /* gameTree maps ObjectID to GameObject */
   private IndexedTree gameTree;

   correspondence
      conc.ROOT = ROOT and
      conc.nodes = gameTree.nodes – { CSR } and
      conc.edges = gameTree.edges –
         { (parent(CSR), CSR) } and
      forall x in conc.nodes,
         conc.order(x) = (
            gameTree.order(x) – 1 if isYoungerSibling(x, CSR);
            gameTree.order(x)     otherwise ) and
      conc.contents = gameTree.contents –
         { (CSR, gameTree.contents(CSR)) }

   public void setObjectProperty(  restores ObjectID obj,
                                   restores Property prop)
      requires obj is_in nodes;
      ensures nodes = #nodes and
         edges = #edges and order = #order and
         contents = #contents override
            { (#obj, [  #contents(#obj).id,
                        #contents(#obj).property union {#prop} ] }
   {
      GameObject rec;
      gameTree.moveBefore(obj);
      gameTree.swapValue(rec);
      rec.addProperty(prop);
      gameTree.swapValue(rec);
   }

   /* other operations */
}
```

Figure 7. GameWorld class

The game world component is implemented with an indexed tree. During game play, each game object is represented by node in the tree. Commands input by the player can potentially update the game world, either by updating the configuration of the tree, or updating the properties of the game objects inside the tree. A portion of the GameWorld class is shown in Figure 7.

The conceptual model of the game world shares the same model variables as the indexed tree, but where the indexed tree is modeled using generic Enums and Objects, the game world uses ObjectID and GameObject types. Also, the game world model does not require a cursor node.

GameWorld contains a single field, the game tree, which is an indexed tree assumed to contain game object identifiers as keys and game objects as values. The correspondence clause, also known as the abstraction relation, describes how to derive the state of the conceptual game world from the state of the internal game tree. The game world is very similar to the game tree except that there is no cursor node and no edge involving the cursor. The order of all younger siblings of the cursor in the game tree are decremented to get their new orders in the game world. The contents are the same except that there is no mapping involving the cursor.

The method setObjectProperty is used in the game to add properties to game objects. It ensures that the structure of the game tree – the nodes, edges, and order – remains unchanged. The new property addition is reflected in the change to the contents variable. The next subsection traces through a particular call to the method, showing how the implementation meets its specification for a particular input.

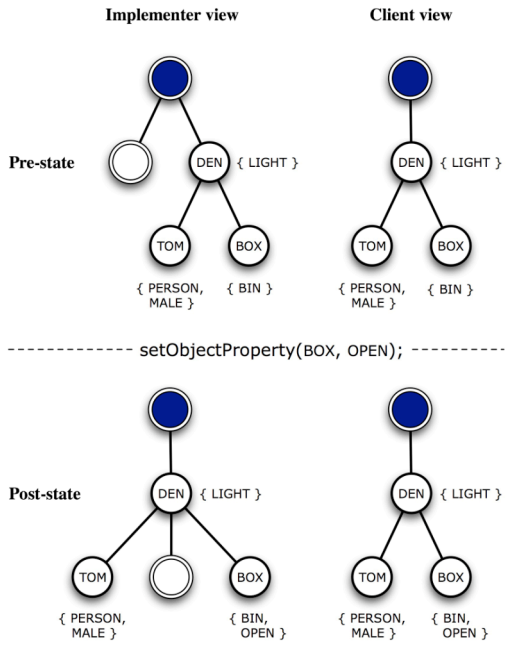


Figure 8. Pre-state and post-state of setObjectProperty call

5.2 Tracing Through a Method

Figure 8 shows a pre-state and the resulting post-state for the call to setObjectProperty(BOX, OPEN). Two views are shown: The implementer view and the client view. The implementer view shows the game tree, which includes a cursor, and the client view shows the game world, in which no cursor is present. Properties for each game object are given next to their corresponding nodes in the trees.

The tracing table in Table 2 shows the state of the program while stepping through the statements in the implementation of the setObjectProperty method. The initial state, state 0, describes the game tree that corresponds to the implementer view of the pre-state in Figure 8. The game tree has five nodes: ROOT, CSR, DEN, TOM, and BOX; and four edges, corresponding to those shown in Figure 8. The order mapping gives the appropriate sibling ranking of the nodes. The contents mapping maps nodes to game objects. In the tracing table, we represent game objects as sets of properties, omitting the redundant identifiers for brevity. The last line of each fact contains the values of the formal parameters and local variables. State 0 begins after the local variable rec is declared.

Table 2. Trace of setObjectProperty (implementer view)

St	Facts
0	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, CSR), (ROOT, DEN), (DEN, TOM), (DEN, BOX) } order = { (ROOT, 1), (CSR, 1), (DEN, 2), (TOM, 1), (BOX, 2) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN}) } obj = BOX **and** prop = OPEN **and** rec = { }
gameTree.moveBefore(obj);	
1	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN}) } obj = BOX **and** prop = OPEN **and** rec = { }
gameTree.swapValue(rec);	
2	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, { }) } obj = BOX **and** prop = OPEN **and** rec = {BIN}
rec.addProperty(prop);	
3	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, { }) } obj = BOX **and** prop = OPEN **and** rec = {BIN, OPEN}
gameTree.swapValue(rec);	
4	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN, OPEN}) } obj = BOX **and** prop = OPEN **and** rec = { }

The statement gameTree.moveBefore(obj) moves the cursor to the position just before BOX. To check if the operation is permissible, we must look at the requires clause of the indexed trees

moveBefore method given in Figure 4. It requires that key (the formal parameter that corresponds to obj) be in the node set. Here, key = BOX, which is in the node set in state 0 just before the call is made, so the precondition is satisfied. To get the facts in state 1, we apply the ensures clause of moveBefore to the facts in state 0.

From the ensures clause for moveBefore we know that the only variables in the program state that change are edges and order. The edge from ROOT to CSR is replaced by an edge from DEN to CSR. BOX is not a younger sibling of CSR, so the order of CSR becomes 2 (the old order of BOX), and BOX's order is incremented. DEN is originally a younger sibling of CSR, so its order is decremented. The order of all other nodes is unchanged. The effects of the other statements are reasoned about similarly.

To verify that the implementation is correct with respect to the specification for this particular start state, we need to translate the first and last states from the implementer view to the client view using the correspondence clause, and then see if they conform to the specification of setObjectProperty. Table 3 is a tracing table for setObjectProperty after this translation. State 0 in Table 3 is derived from state 0 in Table 2 and state 1 is derived from state 4. Applying the ensures clause of setObjectProperty to the facts in state 0 results in the facts in state 1, so the implementation is correct in this instance.

Table 3. Trace of setObjectProperty (client view)

St	Facts
0	**conc**.nodes = { ROOT, DEN, TOM, BOX } **conc**.edges = { (ROOT, DEN), (DEN, TOM), (DEN, BOX) } **conc**.order = { (ROOT, 1), (DEN, 2), (TOM, 1), (BOX, 2) } **conc**.contents = { (ROOT, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN}) }
setObjectProperty(BOX, OPEN);	
1	**conc**.nodes = { ROOT, DEN, TOM, BOX } **conc**.edges = { (ROOT, DEN), (DEN, TOM), (DEN, BOX) } **conc**.order = { (ROOT, 1), (DEN, 2), (TOM, 1), (BOX, 2) } **conc**.contents = { (ROOT, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN, OPEN}) }

6. OTHER DESIGN ISSUES

6.1 Singleton Design Pattern

A design pattern that was used extensively in the Java version of the game, but could not be reproduced in the Tako version was the singleton pattern, whose intent, according to the patterns book of Gamma et al. [11], is to "Ensure that a class only has one instance, and provide a global point of access to it" (p. 127). Note that the statement says nothing about references and nothing about aliasing. However, a typical implementation of the singleton permits aliasing everywhere, as shown in Figure 9.

In this implementation, every singleton variable is a reference to the same object. In practice, however, there is never more than one singleton variable in a class, and there is no benefit from sharing the same object through references over using the object itself. One way to use the same object without aliases is through a global variable. Gamma et. al. note that global variables provide access but criticize global variables for two reasons: They do not prevent multiple instances, and they pollute the global namespace. The first criticism can easily be addressed in Java using the singleton class in Figure 10 in which the constructor is only usable from inside the class itself. The second criticism seems to imply that it is more difficult to reason about the singleton client in Figure 10 than the singleton client in Figure 9 because global variables make reasoning difficult. But this criticism seems odd to us since aliased variables require reasoning about the global heap, a structure as complicated as any variable in a typical program.

The current Tako compiler does not implement static import variables, so a class whose sole purpose was to hold global variables was constructed, and we were disciplined about not declaring instances of, for example, GameWorld, anywhere but inside that class. Clearly this is not a satisfactory solution, so the next version of the Tako compiler will have the ability to implement the singleton as illustrated in Figure 10.

```
class GameWorld {
   GameWorld world = new GameWorld();
   private GameWorld() { /* constructor body */ }
   public static GameWorld getInstance () { return world; }
   /* remainder of class */
}

import GameWorld;
class Resolver {
   GameWorld world = GameWorld.getInstance();
   /* class body uses 'world' variable */
}
```

Figure 9. Typical singleton implementation

```
class GameWorld {
   public static GameWorld world = new GameWorld();
   private GameWorld() { /* constructor body */ }
   /* attributes and methods */
}

import static GameWorld.world;
class Resolver {
   /* class body uses global 'world' variable */
}
```

Figure 10. Singleton implemented with single instance global

6.2 Tako-Java Integration

Tako is syntactically similar to Java, and the current Tako compiler translates to Java, so the current version of Tako is closely tied to the Java language. Given this, we want to ensure that Tako and Java are as compatible as possible. The Tako implementation of the adventure game uses Java Swing components for its graphical user interface. Our initial experience in integrating Java and Tako components helped us come up with a few basic rules, some of which have been applied in the current adventure game, and some of which will have to wait for the next version of the Tako compiler.

A Tako class can use a Java class. Currently, a Tako class simply imports the Java class, but future versions of the compiler should require a special import statement such as "import java". The Tako compiler should translate the Java method call as is. A Java method should not take non-Java arguments. If a Tako variable x is found, the compiler will view it as the function call x.toJava_TypeName() where TypeName is the name of the Java type expected. Obviously, such a function will return a value of type TypeName.

If a programmer wants to write a Java class that uses Tako code, they should write a Java interface and implement that interface with a Tako class. Java methods should have Java parameters and return Java values. Tako classes will need to access Tako types that can convert to and from the Java types needed in the interface methods. For example, if there is a Java method whose signature is String processText(String x), then the Tako Text class should have a method String toJava_String() and a constructor Text(String x).

7. DISCUSSION AND CONCLUSION

A typical Java version of the adventure game might use the composite design pattern to implement the game world. While the composite pattern can be implemented in Tako, the recursive type structures involved raise the possibility that null references will be assigned to variables, which, in general, is something we prefer to avoid. The design of the Tako program uses the game world as a point of centralized access and control for the tree structure and all of its contents. Many object-oriented programmers prefer designs using distributed control rather than centralized control [12]. The Tako language does not preclude designs with distributed control, but formally reasoning about designs can be a challenge. We plan to more explore this topic more thoroughly in future research.

Since all the game objects are stored in a tree like data structure, accessing these objects in Tako meant swapping them out of the structure, examining them, and then swapping them back in. In the Java version, programmers modify a game object through a handle or reference to the object. The swapping in Tako initially lead to errors while inserting them back in the tree as the conceptual cursor position in the tree was unexpectedly modified. The error was easily fixed, but it represents one example of an error that would not occur in Java as the object is never removed in the first place.

In the Java version of the game, the container classes like stacks, hash maps, queues, and lists were already provided by the java.util package. But in the Tako version, these util classes where implemented from scratch using the pointer component. The pointer component was only used in these low level classes; while the higher design level classes did not require the use of the pointer component. This supported our conjecture that the programmer, at higher level of design, can make do without using references.

In this particular program the distinction between object identity and name identity did not play a major role. We used hash maps to store the game objects and each game object had a unique key associated with it. In both the Java and Tako version, it was these unique keys rather than the language dependent object identity that was used for uniquely identifying the objects.

Overall, we found that the paradigm shifts involved were not very difficult to adjust to though they required some alertness in terms of the swapping paradigm. This experience is consistent with that of Hollingsworth et al., who discuss a sizeable commercial application developed in C++ using the Resolve discipline [13]. In their report, they concluded that swapping worked well with most object-oriented techniques.

We would like to implement the adventure game in Java (or perhaps Tako) using the composite design pattern to further explore the benefits and drawbacks of using a tree data structure rather than a composite pattern. Ultimately, we would like to bootstrap the Tako compiler using a design based on formally specified data types. The source code for the current Tako compiler and for the adventure game can be found in the takocompiler project on Sourceforge [14].

8. REFERENCES

[1] Kulczycki, G. and Vasudeo, J. Simplifying Reasoning about Objects with Tako. In *Proceedings of the Specification and Verification of Component-Based Systems 2006* (2006).

[2] Sitaraman, M. and Weide, B. W. Component-Based Software using Resolve. *ACM Software Engineering Notes*, 19, 4 1994), 21-76.

[3] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S. and Hollingsworth, J. E. Reasoning about Software Component Behavior. In *Proceedings of the International Conference on Software Reuse* (2000). Springer-Verlag.

[4] Hogg, J., Lea, D., Wills, A., deChampeaux, D. and Holt, R. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3, 2 1992), 11-16.

[5] Clarke, D. *Object Ownership and Containment*. University of New South Wales, 2001.

[6] Nelson, G. and Rees, G. *The Inform Designer's Manual: 4th Edition*. Dan Sanderson (pub.), 2001.

[7] Roberts, M. J. *TADS - The Text Adventure Development System*. City, 2006.

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. *Introduction to Algorithms: 2nd Edition*. MIT Press, 2003.

[9] Cheon, Y., Leavens, G., Sitaraman, M. and Edwards, S. Model Variables: Cleanly Supporting Abstraction in Design by Contract. *Software - Practice and Experience*, 35, 6 2005), 583-599.

[10] Kulczycki, G. *The Tako Component Library*. City, 2008.

[11] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Adison-Wesley, 2003.

[13] Hollingsworth, J. E., Blankenship, L. and Weide, B. W. Experience Report: Using Resolve/C++ for Commercial Software. In *Proceedings of the International Symposium on Foundations of Software Engineering* (2000).

[14] Kulczycki, G. and Vasudeo, J. *The Tako Compiler Project*. City, 2006.

Integrating Math Units and Proof Checking for Specification and Verification

Hampton Smith
Kim Roche
Murali Sitaraman
Clemson University
School of Computing
Clemson, SC 29634
1-(864)-656-3444
{hamptos | kroche | murali}
@clemson.edu

Joan Krone
Denison University
Mathematics and Computer Science
Granville, OH 43023
1-(740)-587-6484
krone@denison.edu

William F. Ogden
Ohio State University
Computer and Information Science
Columbus, OH 43210
1-(614)-292-1517
ogden@cse.ohio-state.edu

ABSTRACT
A formal system for specification and verification of component-based software must allow extension of the mathematical units available for specification with new mathematical theories just as modern programming languages allow software developers to extend a core collection of data types with new ones by developing reusable software components. These extensions enrich the specification language and lead to simpler specifications. New theory development must also include suitable theorems so that it can be used to support automated proofs of verification conditions (VCs) for correctness arising from annotated implementations. We distinguish between straightforward proofs of VCs and the more nuanced proofs for the theorems in the mathematical units themselves, which often cannot be automated. We explain the need to separate the interface of a mathematical unit (précis) that will be used by software developers and automated provers, from the proof units that contain proofs of theorems. In addition, we describe a mathematician-friendly language for presenting proofs and a proof checker that we have developed to check these proofs.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification—*class invariants*, *correctness proofs*, *formal methods*, and F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*invariants, mechanical verification, pre- and post- conditions*

General Terms
Design, Human Factors, Standardization, Languages, Theory, Verification.

Keywords
Specification, Verification, Proof Checking, Formal Methods

1. INTRODUCTION
The goal of automatically verifying software components with respect to a specification presents a fundamental dilemma. Requiring programmers to engage in a fine level of proof activity is unlikely to lead to wide-spread verification. On the other hand, the limitations of automated theorem proving often require substantial human intervention. Addressing this dilemma is the focus of this paper. We partition the problem of verification to distinguish the roles of software developers from mathematicians and automated provers from proof checkers.

Formal verification ultimately involves the insights of programmers (e.g., specifying invariants), the insights of mathematicians (e.g., discovering non-trivial theorems and furnishing proofs to support them), and the more straightforward task of proving verification conditions of implementation correctness based on these insights. Some verification conditions (VCs) correspond to checking the insights of programmers to eliminate unsoundness that may arise from poor programmer insights. In the scenario we envision, programmers would not be involved in any proving activity beyond documenting their insights. The proving activity would instead be partitioned into two sub-tasks:

1. Proofs of verification conditions to establish the correctness of code.
2. Proofs of supporting theorems from mathematics.

The former would be straightforward and only involve various simplifications so that an automated prover could discharge them without requiring human intervention. The latter would generally require proof steps from mathematicians.

Unlike the proof of VCs arising from code, where the goal is complete automation, the focus of extending the mathematical library is on enabling mathematicians to improve the expressiveness of the theories available to the specification and verification subsystems. It is therefore not necessary that these theorems be automatically verifiable. While a class of theorems can be discharged automatically by automated provers [5, 7, 11, 13], in general proofs of theorems require mathematical insights that cannot be discovered automatically. Obviously, limiting allowable theorems to those that can be automatically proved would in turn limit the class of programs that can be proved. To

address this problem, some current systems (e.g., Isabelle) allow some theorems to be taken for granted without proofs, but clearly this can only be a temporary solution.

To address the sub-problem of proving non-trivial theorems, we have a developed a mathematician-friendly language for writing proofs and a proof checker for checking these proofs. We intend that these proofs will be written by mathematicians, not programmers.

The rest of this paper is organized as follows: In Section 2, we illustrate the need for a verification system to strike a balance between automated theorem proving and mechanically-checked (but user-provided) proofs. In Section 3, we discuss practical consequences of this balance and suggest ways in which the problem may be managed by applying traditional software design tactics such as modularity to the proof subsystem. We support these ideas with examples from the design of our own verification system, RESOLVE. In Section 4, we detail the workings of the proof language and its associated proof checker. In Section 5, we discuss related work and summarize our conclusions.

2. PROOFS OF VCS VS. THEOREMS

To illustrate the distinct issues in proving VCs arising from code and proving theorems in mathematics, we consider a component verification example. In particular, we consider an operation to reverse a given Stack object. A specification and an implementation of the operation are given below in RESOLVE, an integrated specification and programming language [12]. The issues discussed in this paper, however, are language independent.

Specification:

```
Enhancement Flipping_Capability for Stack_Template;
    Operation Flip( updates S: Stack );
        ensures S = Rev(#S);
end Flipping_Capability
```

Code:

```
Realization Obvious_F_C_Realiz for Flipping_Capability
    of Stack_Template;

    Procedure Flip( updates S: Stack );
        Var Next_Entry: Entry;
        Var S_Flipped: Stack;

        While ( Depth( S ) /= 0 )
            changing S, Next_Entry, S_Flipped;
            maintaining #S = Rev(S_Flipped) o S;
            decreasing |S|;
        do
            Pop( Next_Entry, S );
            Push( Next_Entry, S_Flipped );
        end;

        S :=: S_Flipped;
    end Flip;

end Obvious_F_C_Realiz;
```

The specification of Stack_Template on which the Flip enhancement (called an extension operation in other systems) is based imports the mathematical unit String_Theory and conceptualizes a Stack object as a mathematical string. The ensures clause, which defines the behavior of this operation, is used for verification and thus the variables in the clause stand for their mathematical values. In this case, #S refers to the mathematical string that represents the value of the stack S when this operation is called, while S refers to the mathematical string that represents the value of the stack S when this operation exits. Rev is a mathematical function that takes a string and returns it in reverse order.

The While statement in the code for Flip is annotated with three clauses that make the insights of the programmer regarding the correctness of the code explicit. For our purposes, it doesn't matter whether a programmer uses tools to identify and document such assertions (e.g., the work of [3] in identifying loop invariants automatically) or does so herself. The changing clause indicates those variables whose values are permitted to change inside the loop. Implicit is that any variable not mentioned will not change. The maintaining clause provides a loop invariant. The "o" in this line is intended to be read as ∘, the concatenation operator on mathematical strings. The decreasing clause documents the progress metric, i.e., the programmer's rationale for why the loop would terminate.

At the end of the loop, we use the :=: operator, which swaps the values of S and S_Flipped, thus transferring the stack containing the reversed contents to the parameter stack S. The motivation for using swapping and avoiding unnecessary aliasing is the topic of [4].

When the code for Flip is analyzed, the usual syntax-checking and type-checking is performed and, assuming it passes these checks, the code continues to a verifier, which generates VCs that must be proved in order for the code to be considered correct. The VCs generated using the RESOLVE VC generator [9] are shown in an appendix.

The verifier includes a flag to generate Isabelle-friendly assertions (not shown in this paper). Our experience in proving VCs automatically using Isabelle is the topic of [6]. Other example verification benchmarks are given in [15]. All the VCs for the present example can be discharged automatically by the Isabelle prover. Specifically, beyond documenting loop invariants and progress metrics, programmers are not involved at all in verification.

The VCs correspond to checking correctness of programmer-supplied invariants and progress metrics, checking the preconditions of called operations (e.g., Pop), and the postcondition of the operation that is being verified. We discuss the automated verification of one of the VCs to distinguish simplification from theorem proving activities. It is the third VC from the Appendix and it corresponds to the inductive step of establishing the correctness of the invariant. This VC (after removing assumptions that are not necessary) is shown below:

```
((|S| <= Max_Depth) and (S = (Rev(?S_Flipped) o ??S) and
(|??S| /= 0 and ??S = (<?Next_Entry> o ?S))))

======================>

(Rev(?S_Flipped) o ??S) =
(Rev(<?Next_Entry> o ?S_Flipped) o ?S)
```

The VC is an implication. All variables in a VC are mathematical. For example, S is a String of Entries, not a Stack. In this VC, |S| indicates the length of the string S, <X> indicates the string with X as its sole element, and "o" is the concatenation operator on strings.

Variables in the VC prepended with a question mark are verifier-generated and simply represent the values of the variables at different points in the code. So, for instance, S represents the initial value of the stack S, ?S represents its value at the beginning of each loop, and ??S represents its value at the end of each loop.

Automated provers, such as Isabelle, would begin with a substitution in proving this VC:

$(?S\_Flipped^{Rev} \circ ??S) =$
$((<?Next\_Entry> \circ ?S\_Flipped)^{Rev} \circ ?S)$

given

$(?S\_Flipped^{Rev} \circ \mathbf{(<?Next\_Entry> \circ ?S)}) =$
$((<?Next\_Entry> \circ ?S\_Flipped)^{Rev} \circ ?S)$

by substitution

From here, the provers will rely on two important theorems from String_Theory to complete the proof:

Theorem 1:
$\forall \alpha$:String of E, $\forall$x:E, $(\alpha \circ <x>)^{Rev} = (<x> \circ \alpha^{Rev})$.

Theorem 2:
Is_Associative(∘)

Theorem 2 uses the higher order predicate Is_Associative that is defined in a separate math unit named Basic_Function_Properties. This unit defines several other related predicates and is reused by several mathematical units.

Clearly, proving the VC given these theorems is a qualitatively different activity from proving the theorems themselves. Given these theorems, proving the VC is a simple process of repeated substitution. The proofs for the theorems themselves are significantly more involved.

There are certainly automated theorem provers, particularly of the inductive variety, that could provide proofs of Theorems 1 and 2 on their own. ACL2 [1] is one such prover, though it is limited to first order assertions. However, there are many other theorems where automated provers would be unable to make the required logical leap. Indeed, we could imagine writing code that relies on Fermat's Last Theorem for its correctness.

Providing proofs for such theorems, in general, is a process for mathematicians. Programmers cannot be and should not be involved in proving theorems. The simpler task of applying these theorems as part of proving a VC is left for an automated prover. It is our hypothesis that String_Theory can, through careful experimentation and expansion, be fitted with sufficient theorems to make verifying the vast majority of programming concepts based on strings, such as Stacks, Queues, Lists, and others, a task of repeated substitution and thus within the capabilities of a modest automated prover.

Reusing mathematical notions such as strings to specify a wide variety of concepts makes it possible to eliminate the need for Larch-style theories [17] where the theory of Queues is separate from the theory of Stacks, with each different from the theory of Lists.

3. PRÉCIS AND PROOF UNITS

Any code verification system that is complete must provide a mechanism by which arbitrary new theorems can be added; any system that is to be sound must provide a mechanism for providing and checking proofs in support of those theorems. These results in mathematics are reusable in verifying a variety of software artifacts and need to be proved only once. Clearly, developing these proofs is beyond the expertise of typical programmers and should be left to trained mathematicians. This observation suggests a clear division of labor in which programmers are concerned only with immediate details and insights about proving their programs to be correct, whereas mathematicians are involved in proving more general theorems. Like programmers, mechanical provers of VCs need not be concerned with proofs of these theorems.

By linking all programming objects to the mathematical world, mathematical results become applicable in programming contexts. This means, however, that the automated prover is no longer the only entity that needs access to mathematical definitions and results. Software developers also need to be aware of them for use in specifying and verifying software components. However, in both cases they just need to know *what* the results are, but not how they were derived. Therefore, clean, modular, and component-based techniques derived from the world of programming must be applied to the mathematical world of proofs. This is the motivation for separating interfaces of math units (précis) from their corresponding proof units.

Since most readers are familiar with the preliminaries necessary to do proofs with number theory, we use the associativity result on the plus operator on natural numbers as our illustrative example (instead of the string ∘ operator). Many automated provers could, of course, dispatch such a theorem easily. We use it here as an accessible example for when automated proving is not possible.

It is easy to imagine the need for a theorem on the associativity of plus by conceiving of a simple piece of code such as I := (K + L) + M after which, for whatever reason, we need to confirm that I = K + (L + M). Clearly, the validity of this code relies on the associativity of plus on the natural numbers (in the

same way the validity of the Flip code in the previous section relies on the associativity of ∘ on strings.) The précis for Natural_Number_Theory contains the definition of the set N, symbols, such as 0 and suc, and several theorems. We list below one definition and a theorem from this précis:

```
Précis Natural_Number_Theory;
    uses  Basic_Function_Properties,
    Monogenerator_Theory...

    ...

    Inductive Definition on i : N of (a : N) + (b) : N is
        (i) a + 0 = a;
        (ii) a + suc(b) = suc(a + b);

    Theorem N1:  Is_Associative( + );

    ...
end Natural_Number_Theory;
```

A précis is an interface for theory users (both humans and mechanical provers). It provides a summary of the theorems in the theory—everything required to *use* the theorems without any of the details that support the theorems.

This arrangement has obvious analogues to both the header files of C and forms of documentation such as Javadocs. However, unlike C headers, which are primarily intended for use by the compilation system, or Javadocs, which are intended for human consumption, these précis are intended to aid both the verification system and human users. The verification system makes use of proven theorems to verify VCs, mathematicians use them to support new theorems with established ones, and programmers use them to to better tailor their specifications to the available body of mathematical truth. None of these entities need be concerned with the details of supporting proofs. The strict separation of précis from proof unit, enforced by the system, ensures that both documents are always available and synchronized.

The proof for N1 is found in the proof unit Natural_Number_Theory_Proofs. It relies on the definition of a natural number above and reads as follows:

```
Proof unit Natural_Number_Theory_Proofs for
        Natural_Number_Theory;
        Uses ...

Proof of Theorem N1:

Goal for all k, m, n: N, k + (m + n) = (k + m) + n;
Def S1: Powerset(N) =
    { n: N, for all k, m: N, k + (m + n) = (k + m) + n };
Goal S1 = N;
Goal 0 is_in S1;
Goal for all n: S1, suc(n) is_in S1;
Goal for all n: S1, if n is_in S1 then suc(n) is_in S1;
(Base_case) Goal 0 is_in S1;
Goal for all k, m: N, k + (m + 0) = (k + m) + 0;
Goal for all k, m: N, if k is_in N and m is_in N then
    k + (m + 0) = (k + m) + 0;
Supposition k, m: N;
    Goal k + (m + 0) = (k + m) + 0;
        k + (m + 0) = k + m
                                    by (i) of Definition +;
        k + m = (k + m) + 0
                                    by (i) of Definition  +;
Deduction if k is_in N and m is_in N then
    k + (m + 0) = (k + m) + 0;
[ZeroAssociativity] For all k: N, for all m: N,
    k + (m + 0) = (k + m) + 0
                                by universal generalization;
[ZeroInS1] 0 is_in S1
                                    by ZeroAssociativity;
(Inductive_case) Goal for all n: N, suc(n) is_in S1;
Goal for all n: N, if n is_in S1 then suc(n) is_in S1;
Supposition n: S1;
    [InductiveSupposition] For all k, m: N,
        k + (m + n) = (k + m) + n
                                            by Definition S1;
    Goal suc(n) is_in S1;
    Goal for all k, m: N,
        k + (m + suc(n)) = (k + m) + suc(n);
    Goal for all k, m: N,
        if k is_in N and m is_in N then
        k + (m + suc(n)) = (k + m) + suc(n);
    Supposition k, m: N;
        Goal k + (m + suc(n)) = (k + m) + suc(n);
        k + (m + suc(n)) = k + suc(m + n)
                                    by (ii) of Definition +;
        (k + suc(m + n)) = suc(k + (m + n))
                                    by (ii) of Definition +;
        suc(k + (m + n)) = suc((k + m) + n)
                                by InductiveSupposition;
        suc((k + m) + n) = (k + m) + suc(n)
                                    by (ii) of Definition +;
    Deduction if k is_in N and m is_in N then
        k + (m + suc(n)) = (k + m) + suc(n);
    [SucNAssociativity] For all k, m: N,
        k + (m + suc(n)) = (k + m) + suc(n)
                                by universal generalization;
    suc(n) is_in S1
                                    by SucNAssociativity;
Deduction if n is_in S1 then suc(n) is_in S1;
for all n: N, suc(n) is_in S1
                                by universal generalization;
0 is_in S1 and (for all n: N, suc(n) is_in S1)
                                    by ZeroInS1 & and rule;
N = S1
                        by Definition Monogeneric_Pty_3 &
                                            modus ponens;
For all k, m, n: N, k + (m + n) = (k + m) + n
                    by Definition S1 & universal generalization;
Is_Associative( + )
                            by Definition Is_Associative( + );
QED

end Natural_Number_Theory_Proofs;
```

The proof language uses a syntax that mimics the traditional style of a mathematical proof. Provers such as Isabelle use a programming language-like syntax for expressing mathematics to enable ease of automation. Unfortunately, this very reason may make it less intuitive for traditional mathematicians. Because we have drawn a clear separation between automated verification of VCs and proof checking for theorems, we can use a language for writing proofs that is more intuitive for mathematical users. To this end, "Goals" are comments to state what the proof will try to do next; "Supposition/Deduction" pairs provide a mechanism for establishing implications; "definitions" can be introduced on the fly; and the "by" keyword introduces the rationale for the next step. A line of the proof can be given a label in square brackets for future reference.

This proof begins by stating a number of *Goal*s. The proof establishes a set, S1, which is defined to be the power set of N, for which the property of associativity already holds. The proof then proceeds with an induction over the natural numbers to prove that the set S1 is the same set as N. The base case of this induction is to prove that the natural number 0 is in S1, which is accomplished by using the identity property inherent in the definition of + to show that when adding zero, at least, + is associative, and thus 0 is also in S1

To see how straight forward the task of mechanization by the proof checker is, consider the italicized line that makes use of the "and" rule.

0 is_in S1 and (for all n: N, suc(n) is_in S1)
by ZeroInS1 & and rule;

It is simply the conjunct of the assertion labeled *ZeroInS1* with the assertion in the previous line.

A basic tenant of this proof checker is to approach a minimal basis of justifications to explain the transitions from step to step within a proof. These justifications act in concert with references to provide the rationale for a single step. A reference names one of the following kinds of entities:

- **Lemmas**, which are found in math précis or locally in a proof unit;
- **Theorems**, which are found in math précis;
- **Suppositions** that were established earlier in the proof;
- **Labels** that were given earlier in the proof;
- **Definitions/Corollaries**, which may be found inside a theory or defined earlier in the proof.

The available justifications are split into three groups based on the number of references they act on. The justifications requiring two references are as follows:

- Modus ponens
- And rule
- Contradiction
- Alternative elimination
- Common conclusion

The justifications requiring one reference are:

- Equality
- Reductio ad absurdam
- Existential generalization
- Or rule
- Conjunct elimination
- Quantifier distribution
- Definition ∃!
- Universal instantiation
- Existential instantiation

Finally, the only justification requiring no references is:

- Excluded middle

4.PROOF CHECKER

Each justification has a well defined meaning that allows the proof-checker to determine if it is valid. Consider the following example of the semantics of *modus ponens*:

Γ, δ ⊢ [Label] B
by [Reference2,] Reference1 & modus ponens
where {A, A → B} ⊆
Extract{[Reference2,] Reference1}, Γ, δ}

Γ represents the theories (with comprising theorems) currently in scope of the proof; δ represents the derivation of the proof so far; ⊢ is an operator indicating that the following application of a justification is valid; *A* and *B* are simply mathematical expressions; and the Extract function returns the set of mathematical expressions that have been assumed so far that correspond to the given names within either Γ or δ, or that was assumed in the immediately preceding line of the proof. Square brackets indicate an optional part.

So, overall this line is to be read, "A justification by *modus ponens* is permitted for establishing *B* if the given references and the expression of the immediately preceding line are sufficient to establish, from Γ and δ, that (*A* → *B*) and (*A*). In the future, the expression *B* may itself be referenced as *Label*."

As another example, here is the semantics of *alternative elimination*:

Γ, δ ⊢ [Label] A
by [Reference2,] Reference1 & alternative elimination
where {B} ⊆ Extract{[Reference2,] Reference1}, Γ, δ}
and {A or B, B or A} ∩
Extract{[Reference2,] Reference1}, Γ, δ} ≠ ∅

This is to be read, "A justification by *alternative elimination* is permitted for establishing *A* if the given references and the expression of the immediately preceding line are sufficient to establish, from Γ and δ, ¬*B*, and at least one of (*A* or *B*) or (*B* or *A*) can be established in the same way. In the future, the expression *A* may itself be referenced as *Label*."

The current version of the proof checker is able to verify valid proofs (including the proof of associativity provided in Section 3), though higher-order theorems such as Is_Associative are not yet implemented. In addition, it is able to recognize and produce appropriate error messages for attempts to apply faulty justifications. We provide two examples of simple proofs containing errors in logic and the output of the proof checker when run on each.

First consider this working example:

Corollary Identity: a : N and a + 0 = a;

Proof of Theorem Nothing:
Supposition k, m: N;
(k + m) + 0 = k + m
by Corollary Identity & equality;
Deduction if k is_in N and m is_in N then
(k + m) + 0 = k + m;
QED

This proof simply establishes that given the identity property of addition on natural numbers, if two numbers, *k* and *m*, are natural numbers, then *(k + m) + 0 = (k + m)*.

Now consider an invalid application of the Identity Corollary:

```
Supposition k, m: N;

    (k + m) + 0 = m + 0 by Corollary Identity & equality;

Deduction if k is_in N and m is_in N then
    (k + m) + 0 = k + m;
```

When run through the proof-checker, this produces the following output:

```
Error: Simple.mt(10):
Could not apply substitution to the justified expression.
    (k + m) + 0 = m + 0 by Corollary Identity & equality;
```

Next, consider an invalid choice of justification to an otherwise valid step:

```
Supposition k, m: N;
    (k + m) + 0 = k + m by Corollary Identity & or rule;
Deduction if k is_in N and m is_in N then
    (k + m) + 0 = k + m;
```

When run through the proof-checker, this produces the following output:

```
Error: Simple.mt(10):
Could not apply the rule Or Rule to the proof expression.
        (k + m) + 0 = k + m by Corollary Identity & or rule;
```

5. RELATED WORK AND CONCLUSIONS

5.1 Isabelle

Isabelle is a proof assistant implemented in Standard ML and based on the specification language *Isar* [13, 16]. Its focus is on interactive proof development. It is also able to complete proofs automatically. Unlike earlier versions that closely resembled ML syntax, more recent versions have begun to put more of an emphasis on human readability of proofs. Even with these improvements, proofs contain artifacts of programming languages. For example, consider the following (trivial) proof that A and B → B and A where A and B are complicated expressions (called large_A and large_B in the proof) modified from [14]:

```
lemma assumes AB: "large_A ∧ large_B"
    shows "large_B ∧ large_A" ( is "?B ∧ ?A")
    using AB
proof
    assume "?A" "?B" show ?thesis ..
qed
```

While penetrable, it is harder to follow for those whose background is purely mathematical. Also, Isabelle proofs include statements to help ease automation, often interspersed with steps of the proof itself. By separating proofs that are merely checked from those that are totally automated, this difficult can be avoided.

Isabelle differs from RESOLVE as a language for proofs primarily with respect to its syntax, which maintains a programming language flavor. Another difference is that Isabelle provides no specific support for syntactically separating theorems from their proofs, though tools are provided for auto-generating documentation that serves much the same purpose. Also, Isabelle permits the bodies of proofs to be elided using a "sorry" command, which may allow unsound theorems to be introduced into the system.

5.2 Coq

Both Coq and RESOLVE share an emphasis on a small but extensible logical core and a specification language tailored for the tool itself (in the case of Coq, this language is called *Gallina*) [5]. Coq has limited automatic proving capabilities and a syntax more reminiscent of a programming language than a mathematical proof. As an example, consider a proof in Coq that A and B → B and A modified from [5]:

```
Variables A B C : Prop.

Lemma and_commutative : (A /\ B) -> (B /\ A).
    intro.
    elim H.
    split.
    exact H1.
    exact H0.
Save.
```

As with Isabelle, there is no explicit syntactic mechanism for separating theorems from their proofs; though, again, tools exist to automatically generate documentation. Also, like Isabelle, Coq provides a "trust me" command, which allows a proof to be elided.

5.3 PVS

PVS exists somewhere between a proof assistant and a theorem prover [2, 10, 11]. It uses a library of definitions and theorems to support the SMT solver *Yices* for the automatic verification of arithmetic expressions and equalities. Unlike RESOLVE, PVS has almost no emphasis on human-readable proofs. PVS's type checking system occasionally defers to the proof-checker to resolve ambiguous proof conditions. This contrasts with RESOLVE, where code, specifications, and proofs must all pass type checking before moving on to verification.

5.4 Nuprl

Like Isabelle, Nuprl is based on ML. Unlike RESOLVE, it does not perform type-checking on proofs [7, 8]. Nuprl relies on a built-in set of theories such as integers, function, and sets, which can only be extended by the use of tuples, unions, and lists. This contrasts with RESOLVE where only a minimal set of theories is provided (namely Boolean theory and a small portion of Set theory) from which other theories are built.

5.5 Conclusions

Software verification is a challenging problem. To address it effectively, a formal verification system that includes a verifying compiler needs to bring together the insights of programmers and mathematicians with advances in prover technology for

mechanizing straightforward proofs, and proof checking for non-trivial theorems. Here we have presented a framework for addressing this challenge along with a summary of our efforts in proof checking. We plan much more experimentation with the ideas and tools presented here in order to make progress toward a sound and complete verification system.

6. ACKNOWLEDGMENTS

This research is funded in part by NSF grants DUE-0633506, DMS-0701187, DMS-0811748, and CCF-0811748. We thank the members of the RESOLVE/Reusable Software Research Group at Clemson and Ohio State for discussions on the topics presented in this paper. Special thanks are due to Jeremy Avigad at CMU, and Harvey Friedman and Bruce Weide at Ohio State. We thank the referees whose comments have helped improve the paper. We also thank the attendees of the RESOLVE meeting held at Clemson in 2007 when Kim Roche presented a draft version of these ideas.

7. APPENDIX

7.1 Stack Specification

```
Concept Stack_Template(type Entry; evaluates Max_Depth:
Integer);
   uses Std_Integer_Fac, String_Theory;
   requires Max_Depth > 0;

   Type Family Stack is modeled by Str(Entry);
      exemplar S;
      constraint |S| <= |Max_Depth|;
      initialization ensures S = empty_string;

   Operation Push(alters E: Entry; updates S: Stack);
      requires 1 + |S| <= Max_Depth;
      ensures  S = <#E> o #S;

   Operation Pop(replaces R: Entry; updates S: Stack);
      requires |S| > 0;
      ensures #S = <R> o S;

   Operation Depth(restores S: Stack): Integer;
      ensures Depth = (|S|);

   Operation Rem_Capacity(restores S: Stack): Integer;
      ensures Rem_Capacity = (Max_Depth - |S|);

   Operation Clear(clears S: Stack);

end Stack_Template;
```

The concept Stack_Template is parameterized by a type, Entry, comparable to a generic in Java, and a Max_Depth that ensures each stack never becomes deeper than some capacity.

A type, Stack, is introduced, which is modeled on a mathematical string of Entrys. The constraints clause introduces a class invariant: the depth of a stack may never exceed Max_Depth. The initialization ensures clause guarantees that all implementations of Stack will ensure that new Stacks begin empty.

Next comes a list of the usual operations on Stacks. Each operation has a requires clause, which states the operation's pre-condition; and an ensures clause, which states its post-condition. In the ensures clause, a variable like S refers to the outgoing value of S while #S refers to the initial, incoming value of S. In addition to a type, each parameter in an operation has a *parameter passing mode*, such as alters or updates. These modes make certain assurances about the way in which a given parameter will be used. For instance, the alters mode indicates that the incoming value of the parameter is meaningful (and thus that variable may appear in the requires clause), but that the outgoing value of that parameter is undefined (and thus referring to the outgoing value in the ensures clause is illegal.) Updates indicates that both the incoming and outgoing values are defined. The others are similar.

As an example, the Pop operation takes a Stack, S, and an Entry, R, into which to pop the top entry. The requires clause states that there must be at least one Entry on S, and the ensures clause states that when we have finished, prepending R onto the final value of S will have the same value as the initial value of S.

7.2 VCs Resulting from Obvious_F_C_Realiz

```
Free Variables: Max_Depth:*Z, min_int:*Z, max_int:*Z,
S:*Str(*Entry), ?S:*Str(*Entry), ?Next_Entry:*Entry, ?
S_Reversed:*Str(*Entry), Next_Entry:*Entry,
S_Reversed:*Str(*Entry)

((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0) and
(0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|))
======================>
S = (Rev(empty_string) o S)

(((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0)
and (0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|)) and (S = (Rev(?S_Rev) o ?S) and |?S| /= 0))
======================>
((1 + |?S_Reversed|) <= Max_Depth)

(((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0)
and (0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|)) and (S = (Rev(?S_Reversed) o ?S) and |?S| /
= 0))
======================>
(Rev(?S_Reversed) o ?S) = (Rev((<?Next_Entry> o ?
S_Reversed)) o ?S)

(((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0)
and (0 < max_int) and (Max_Depth > 0))) and (|S| <= |
Max_Depth|)) and (S = (Rev(?S_Reversed) o ?S) and |?S| /
= 0))
======================>
(|?S| < |?S|)

((((min_int <= 0) and (0 < max_int)) and ((min_int <= 0) and
(0 < max_int) and (Max_Depth > 0))) and ((|S| <= |
Max_Depth|) and (S = (Rev(?S_Reversed) o ?S) and |?S| =
0)))
======================>
?S_Reversed = Rev((Rev(?S_Reversed) o ?S))
```

8. REFERENCES

[1] "ACL2 Version 3.4: The User's Manual." http://www.cs.utexas.edu/users/moore/acl2/v3-4/acl2-doc.html#User%27s-Manual

[2] B. Dutertre and L. de Moura, "The Yices SMT Solver," August 2006, http://yices.csl.sri.com/documentation.shtml./

[3] M. D. Ernst. Dynamically discovering likely program invariants. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[4] D. E. Harms and B. W. Weide, Copying and Swapping: Influences on the Design of Reusable Software Components, *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, May 1991, pp. 424 - 435.

[5] G. Huet, G. Kahn, and C. Paulin-Mohring, "The Coq Proof Assistant: A Tutorial." INRIA, 2004, pp. 3-18; 45-47.

[6] H. Kirschenbaum, K. Harton, and M. Sitaraman, A Case Study in Automated Verification, *Proceedings of CAV/AFM Workshop*, Princeton, NJ, July 2008.

[7] PRL Project, "The Nuprl Book," September 1995, http://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html

[8] PRL Project, "Nuprl Basics – Nuprl Primitives," September 2003, http://www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/.

[9] RESOLVE Compiler and Verifier. http://www.cs.clemson.edu/~resolve/compiler-verifier.html.

[10] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, "PVS Language Reference: Version 2.4." Menlo Park, CA: SRI International, 2001.

[11] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, "PVS Prover Guide: Version 2.4." Menlo Park, CA: SRI International, 2001, pp. 1-24; 103-110.

[12] M. Sitaraman, and B. Weide, eds., Special Feature: Component-Based Software Using RESOLVE, *Software Engineering Notes 19*, 4 (October 1994), 21-22.

[13] T. Nipkow, L. C. Paulson, M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic." New York: Springer-Verlag, 2008, Sections 1.1, 1.2, and 2.3.

[14] T. Nipkow. "A Tutorial Introduction to Structured Isar Proofs," http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-overview.pdf.

[15] B. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum and D. Frazier. Incremental Benchmarks for Software Verification Tools and Techniques. *Proceedings of VSTTE 2008*, Toronto, CA, Oct 2008, to appear.

[16] M. Wenzel, "Isabelle/Isar: Reference Manual", June 2008, www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-ref.pdf. Section 4.4.

[17] J. M. Wing. Using Larch to specify Avalon/C++ objects. *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990, pp. 1076-1088.

Using Isabelle to Help Verify Code That Uses Abstract Data Types

Jason Kirschenbaum
The Ohio State University
Columbus, OH 43210, USA
kirschen@cse.ohio-state.edu

Bruce M. Adcock
The Ohio State University
Columbus, OH 43210, USA
adcockb@cse.ohio-state.edu

Derek Bronish
The Ohio State University
Columbus, OH 43210, USA
bronish@cse.ohio-state.edu

Paolo Bucci
The Ohio State University
Columbus, OH 43210, USA
bucci@cse.ohio-state.edu

Bruce W. Weide
The Ohio State University
Columbus, OH 43210, USA
weide@cse.ohio-state.edu

ABSTRACT

Verification of programs that use abstract data types (ADTs) is an important piece of the grand challenge of verified software. It is our position that an interactive proof assistant, such as Isabelle, used in a fully automated mode, can be an effective, extensible proof engine for use in the modular verification of software. As technical justification for this position, we describe the modular verification of two implementations of an extension to a queue ADT. One implementation is recursive, while the other is iterative and relies on a stack ADT. The correctness of the implementations is proved by Isabelle automatically, using specification theories from the Resolve mathematical library imported into Isabelle. Isabelle's viability as a general-purpose VC prover is also discussed.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness Proofs*

General Terms

Languages, tools

Keywords

Verification, Isabelle, formal methods, reuse

1. INTRODUCTION

The grand challenge for computer scientists to produce a verifying compiler, recently reissued by Hoare [18], has been a goal of the community since as early as 1967 [14]. A verifying compiler parses and compiles code, generates verification conditions (VCs) whose validity implies the correctness of the code, and proves or refutes those VCs via automated reasoning methods. Thus any object code generated by the compiler is certified correct relative to a specification of the intended behavior the code implements. Figure 1 diagrams how such a process might work. Programmers write assertive code in an attempt to meet the given specifications. A VC generator then processes the code and the specifications, supplemented with relevant mathematical theories and definitions. An automated theorem prover is fed the resulting VCs, along with the theories, definitions and previously established mathematical results. The theorem prover may rely on specialized decision procedures and/or general-purpose reasoning methods. In the case that the VC is proven, a proof checker can be invoked to confirm the proof to rule out possible errors in the prover, which is both complex and generally treated as a black box.

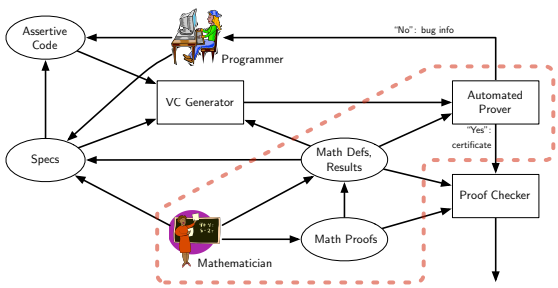


Figure 1: Envisioned framework for verified software

In this paper, we are concerned with the *automatic* proof of VCs, focusing on the region delineated by the dotted line in Fig. 1. More specifically, this paper focuses on an *extension* of an ADT, and does not investigate the verification of ADT implementations themselves. The extension contains the mathematical specification of a new operation for the ADT, along with one of many possible implementations. In accordance with the usual object-oriented design princi-

ples, the implementation does not use knowledge about the internal data representation of the ADT. Therefore, the operation can be verified by examining only the specifications of any operations called in the code, and the specification that the code purports to implement. Any verifying compiler should only have to perform the proof of the correctness of this code once. This is the basic tenet of modular verification [24].

It is our position that the proof assistant Isabelle [21] can function as an effective, extensible tool to perform the automated verification of software, in a manner similar to SMT solvers [11, 10]. Moreover, we can factor out the mathematical theories needed from the proof engine used for the VC proofs.

In order to justify our position, we present two examples of automated verification for extensions of abstract data types. These examples are from a recently proposed set of benchmarks for automated verification [29]. The mathematical theory involved is that of strings. Next, the advantages of our approach are discussed. Specifically, we illustrate how the disciplined use of a small set of rich mathematical models used in model-based specifications permits the expression of required loop invariants, even where two different ADTs are involved (in this case, stacks and queues). Finally, we turn our attention to the issues that naturally arise as we continue to expand the power of the mathematical theories to specify additional ADTs and extensions using the same approach.

We use the interactive proof assistant Isabelle [21] to prove the VCs. The choice to use an interactive proof assistant rather than a specialized decision procedure for the relevant mathematical theories was motivated by concerns about how the approach scales upwards. The primary drawback to customized decision procedures is that a new one needs to be formulated and proven correct for every new mathematical entity that the code can realize. Of course, interactive proof assistants have their downsides as well, for example the question of whether their failure to establish a result is due to an actual deficiency in the code to be verified, or a limitation in the prover itself. While, in the long term, a hybrid approach may be used in practice, we are interested in the research questions involved in comparing these two approaches.

Our choice of Isabelle, rather than another interactive proof assistant such as PVS [23] or COQ [1], is motivated by several factors. First, one of the members of our team is an expert in Isabelle, which eases the learning curve to use Isabelle (similarly any interactive proof assistant). The second is the popularity of Isabelle/HOL in the area of software verification [2, 28, 5, 7]. Finally, Isabelle/HOL's ability to add new lemmas to the automated proof tactics was also a consideration.

The language of the specifications and implementations is a dialect of Resolve [12, 22, 4]. This language provides support for the separation of component specification and implementations, with constructs admitted to the language only when they allow proof rules that support modular verification. The language provides a *value* semantics mental model of ADTs. There are no references; aliasing cannot occur [27]. The specifications are model-based, and are manifested as `requires` and `ensures` clauses on each of the operations, thus creating "operation contracts." Built-in to Resolve are several fundamental mathematical theories that are powerful enough to specify most ADTs. When necessary, additional theories can be defined [12]. The number of theories used is, however, intended and expected to remain small, and Resolve includes the core theories of strings, integers, real numbers, sets, multisets, trees, binary trees, tuples, and booleans.

The specifications shown here are similar to those found in other specification languages. The queue ADT example presented here could be specified in a similar manner in JML [20]. However, the presence of aliasing and other features in Java significantly complicates the specifications; for example, the specification of a `Stack` in JML [20] is far more complex than the Resolve specification of `Stack` (presented in the next section).

Section 2 describes the contracts and implementations of the examples studied, along with the relevant mathematical theory. Section 3 discusses the most interesting VCs necessary for the correctness of the implementations. Section 4 summarizes our experiences with proving correctness of VCs involving string models. Section 5 discusses related work. Finally, Section 6 concludes with possible future directions.

2. SPECIFICATION AND IMPLEMENTATION OF THE QUEUE REVERSE PROCEDURE

2.1 Resolve String Theory

The contract specifications of the ADT for queue and its reverse extension are in terms of mathematical strings (Fig. 2). These mathematical strings are used as a mathematical model of the behavior of the queue ADT similar to what can be done using sequences in JML [6]. We emphasize that specifications and code are two different entities; specifications are statements in mathematics that may have multiple valid realizations in code, and are not themselves executable. [1]

Informally, strings over a given type *obj* are intended to have a model that is exactly the elements of obj^*, where * is the Kleene star. However, a theory such as the one in Fig. 2 is self-contained and might have other models; that is, a theory is defined not by exhibiting a particular model, but rather by its axioms. Of course, it is important to know that such a model exists and that we are not describing an inconsistent theory. We briefly describe the process of ensuring that this theory is consistent in Section 2.4.

A few functions are defined for strings: concatenation, length, and reverse. In Fig. 2, we state some simple lemmas without proofs. The proofs of these lemmas follow easily from the axioms and definitions. Once proved, these lemmas can be used in proofs of VCs, exactly like the axioms and definitions.

[1] We write Resolve mathematical theories in a mathematical notation, as shown here. Isabelle proof scripts are shown in the ASCII equivalent. This choice explicitly delineates the proofs from the underlying mathematical theories.

String Type Signature

$string \stackrel{\text{def}}{\equiv} string(obj)$
$\Lambda : string$
$\text{ext} : string \times obj \longrightarrow string$

String Axioms

1. $\text{ext}(s, x) \neq \Lambda$
2. $\text{ext}(s_1, x_1) = \text{ext}(s_2, x_2) \Rightarrow s_1 = s_2 \wedge x_1 = x_2$
3. $\forall S \in \mathcal{P}(string) : (\Lambda \in S \wedge \forall x, s : (s \in S \Rightarrow \text{ext}(s, x) \in S)) \Rightarrow S = string$

Function Definitions

1. $\langle \_ \rangle : obj \longrightarrow string \stackrel{\text{def}}{\equiv} \langle x \rangle = \text{ext}(\Lambda, x)$
2. $|\_| : string \longrightarrow \mathbb{N} \stackrel{\text{def}}{\equiv} |\Lambda| = 0 \wedge |\text{ext}(s, x)| = |s| + 1$
3. $* : string \times string \longrightarrow string \stackrel{\text{def}}{\equiv} (s * \Lambda = s) \wedge (s_1 * \text{ext}(s_2, x) = \text{ext}(s_1 * s_2, x))$
4. $\text{reverse} : string \longrightarrow string \stackrel{\text{def}}{\equiv} \text{reverse}(\Lambda) = \Lambda \wedge \text{reverse}(\text{ext}(s, x)) = \langle x \rangle * \text{reverse}(s)$

Useful Lemmas

1. lemma EmptyNotSingle: $\Lambda \neq \langle x \rangle$
2. lemma IdofEmpty : $\Lambda * \alpha = \alpha$
3. lemma LenofSingle : $|\langle x \rangle| = 1$
4. lemma LenofCat : $|\alpha * \beta| = |\alpha| + |\beta|$
5. lemma AssocCat : $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$
6. lemma ReverseofReverse : $\text{reverse}(\text{reverse}(\alpha)) = \alpha$
7. lemma ReverseofCat : $\text{reverse}(\alpha * \beta) = \text{reverse}(\beta) * \text{reverse}(\alpha)$
8. lemma LenofReverse: $|\text{reverse}(\alpha)| = |\alpha|$

Figure 2: String Theory

2.2 Specifications

The queue reverse extension uses a queue ADT, which is specified in the `QueueTemplate` component. Figure 3 shows the contract with preconditions and postconditions on each operation written via `requires` and `ensures`, respectively. The mathematical model of a queue is a string of items, and its initial value is Λ which is represented in ASCII as `empty_string`. This component is parametrized by the type of items in the queues. The usual queue operations are all specified in terms of this string model. The parameter modes used include `updates`, `clears`, `replaces`, and `restores`. The `updates` mode indicates that the parameter may be modified by the procedure in accordance with the `ensures` clause. The `clears` mode means the parameter has an initial value for its type upon return. The `replaces` mode indicates that the corresponding argument may be modified but that the incoming value of the parameter has no effect on the behavior of the operation. Finally, the `restores` mode means that the incoming and outgoing values of the parameter are equal. Note that the semantics of the `clears` and `restores` parameter modes each induces a proof obligation that must be discharged in order for an implementation to be verified. In `ensures` clauses, the `#` indicates the old value of a variable. In the `Queue` type declaration, the scope of `exemplar q` is just the `initialization ensures` clause; it introduces a name for an arbitrary object of the new type.

```
contract QueueTemplate (type Item)

    math subtype QUEUE_MODEL is string of Item

    type Queue is modeled by QUEUE_MODEL
        exemplar q
        initialization ensures
            q = empty_string

    procedure Enqueue (updates q: Queue,
                       clears x: Item)
        ensures
            q = #q * <#x>

    procedure Dequeue (updates q: Queue,
                       replaces x: Item)
        requires
            q /= empty_string
        ensures
            #q = <x> * q

    function IsEmpty (restores q: Queue): control
        ensures
            IsEmpty = (q = empty_string)

end QueueTemplate
```

Figure 3: Queue ADT Specification

Finally, the parameter passing for each of the operations is performed via swapping [16]. In the absence of repeated arguments which are ruled out by the syntax of Resolve in this dialect, this method of parameter passing is equivalent in behavior to pass by reference.

```
contract QueueReverse enhances QueueTemplate

    procedure Reverse (updates q: Queue)
        ensures
            q = reverse (#q)

end QueueReverse
```

Figure 4: Queue Reverse Specification

2.3 Recursive and Iterative Realizations

The functionality described in `QueueTemplate` is extended by `QueueReverse`, which specifies a new procedure operation: `Reverse`. This is shown in Fig. 4. The specification uses the *mathematical* function `reverse`, from the string theory of Fig. 2.

An implementation of the `Reverse` procedure can be done in at least two different ways: recursively, or iteratively using a stack. The stack ADT is specified in a way similar to queue, again using strings as the mathematical model. The `Push` and `Pop` operations both work on the "left" end of the string that models a stack. The stack provides LIFO behavior, while the queue provides FIFO behavior. The contract of the stack ADT is needed for the proof of correctness, of course, and is shown in Fig. 5.

The iterative `Reverse` implementation performs the reversal in two steps, as shown in Fig. 6. First, all of the elements of the queue are moved from the queue to a local stack. Then all of the elements of the stack are popped off and placed in the queue. In this implementation, we must verify two loops. A loop includes both a loop invariant (the `maintains` clause) and a loop progress metric (the `decreases` clause, which is used to prove termination). In a loop invariant, `#` denotes

```
contract StackTemplate (type Item)

    math subtype STACK_MODEL is string of Item

    type Stack is modeled by STACK_MODEL
        exemplar s
        initialization ensures
            s = empty_string

    procedure Push (updates s: Stack, clears x: Item)
        ensures
            s = <#x> * #s

    procedure Pop (updates s: Stack, replaces x: Item)
        requires
            s /= empty_string
        ensures
            #s = <x> * s

    function IsEmpty (restores s: Stack): control
        ensures
            IsEmpty = (s = empty_string)

end StackTemplate
```

Figure 5: Stack ADT Specification

the value of the variable just before execution encounters the loop.

```
realization Iterative implements QueueReverse

    facility StackFacility is StackTemplate (Item)

    procedure Reverse (updates q: Queue)
        variable s: Stack
        loop
            maintains reverse(#s) * #q = reverse(s) * q
            decreases |q|
        while not IsEmpty (q) do
            variable x: Item
            Dequeue (q, x)
            Push (s, x)
        end loop
        loop
            maintains #q * #s = q * s
            decreases |s|
        while not IsEmpty (s) do
            variable x : Item
            Pop (s, x)
            Enqueue (q, x)
        end loop
    end Reverse

end Iterative
```

Figure 6: Iterative Queue Reverse Implementation

The recursive implementation of the `Reverse` procedure is demonstrated in Fig. 7. This implementation removes the first element from the queue, recursively reverses the rest of it, then enqueues the removed element. Since this procedure is recursive, we must provide a metric that decreases in each recursive call to `Reverse` in order to prove total correctness.

These implementations cover several of the standard features of any imperative programming language: (recursive and non-recursive) calls, loops and conditional control structures, and the use of ADTs. The examples show how our tools and techniques for proving VCs process these common language features.

The lemmas needed to prove the correctness of the two implementations of `Reverse` are few, as seen in Fig. 2. We

```
realization Recursive implements QueueReverse

    procedure Reverse (updates q: Queue)
        decreases |q|
        if not IsEmpty (q) then
            variable x: Item
            Dequeue (q, x)
            Reverse (q)
            Enqueue (q, x)
        end if
    end Reverse

end Recursive
```

Figure 7: Recursive Queue Reverse Implementation

do not need to include two mathematical theories, one for stacks and one for queues; string theory is rich enough for both. This mathematical uniformity is especially advantageous for the iterative version. It would be tricky to write both loop invariants without it.

2.4 String Theory in Isabelle

Isabelle [21] is an automated proof assistant, meaning it is capable of checking a proof that a user directs. Automated proof assistants can also perform many of the tedious steps needed to produce a proof; in some cases the proof assistant can produce the proof outright, establishing the goal without human guidance.

More specifically, Isabelle has a simplifier that can be invoked to simplify assumptions and goals of a theorem. Isabelle also includes a classical reasoner that can perform many of the logical inference rules automatically. The proof structure of Isabelle is set up in a manner that mimics natural deduction. Assumptions and a goal are presented and simplifications can apply to both. Rules for applying already-proved lemmas and theorems dictate how the assumptions and goals are modified. One can apply forward reasoning and modify the assumptions, apply backward reasoning and modify the goals, or apply both at the same time.

For convenience, many of the proof methods instantiated with common lemmas are performed in Isabelle via the `auto` and `force` commands, commonly referred to as "tactics." These tactics use both the simplifier and the classical reasoner, the difference being that the `force` method will only succeed or fail, while the `auto` method will return a simplified goal (if possible).

New axiom systems, theories, and theorems can be entered into Isabelle using a built-in meta-level logic. All other axiom systems are implemented on top of this meta-logic. For example, higher order logic (HOL) [21] and ZF set theory [25] are available.

Since string theory is already developed in Resolve, we need only to import that theory into Isabelle. The proofs of the lemmas in string theory can then be factored off from the usage of those lemmas for proving the various VCs. Of course, a theory must also have a witness to the existence of a model that satisfies its axioms. Fortunately, such a model for string theory is readily available, namely the Isabelle `List` type in the HOL theory. More information about the process

used to import the Resolve String theory into Isabelle can be found on the web at `http://www.cse.ohio-state.edu/~kirschen/rsrg/Isabelle.html`

We do not use the plethora of theories available in Isabelle for the automatic proof of VCs, but rather use Isabelle's proof engine along with *only* the theories already developed for Resolve specifications. By doing so, we are not tied down to any particular proof assistant or theorem proving tool. As long as the tool has an expressive enough proof language, and allows users to add new simplification and proof rules, then it might be used for our purposes.

3. VERIFICATION AND RESULTS

```
theory RecursiveQueueReverse_Reverse

imports Main String

begin
...
lemma 4:

"[|
    is_initial((x_2::'obj)) ;
    is_initial((x_5::'obj)) ;
    ~<(x_3::'obj)> o (q_3::'obj string) = empty_string
 |]
 ==>
    reverse(q_3) o <x_3> = reverse((<x_3> o q_3))"

apply (((simp only: simp_thms),clarify?)+)?

apply (force+)?

done
...
end
```

Figure 8: A key VC for the verification of the recursive implementation of Reverse

The method of generating VCs [17, 27] is known to be both sound and relatively complete (i.e., relative to the completeness of the mathematics used in the specifications). This method of generating VCs is quite similar to a method described by Barnett *et al.* [3]. At a high level, the generation of VCs involves processing the realization's code (accumulating facts from the `ensures` clauses of each procedure used) while taking control structure (conditionals and loops) into account. A new VC is generated for each `requires` clause of a called procedure or function that is not syntactically "true," at each loop invariant and recursive call, and at the end of the operation body.

Lemma #4 (state index: 6, ensures clause)

$$
\begin{array}{ll}
 & \text{is\_initial}(x_2) \\
\wedge & \text{is\_initial}(x_5) \\
\wedge & \langle x_3 \rangle * q_3 \neq \Lambda \\
\Rightarrow & \text{reverse}(q_3) * \langle x_3 \rangle = \text{reverse}(\langle x_3 \rangle * q_3)
\end{array}
$$

Figure 9: Human readable version of Fig. 8

The VCs are intended not only to be mathematically precise, but also human-readable. At a high level, between each pair of statements in the implementation's code a new subscript is created for each variable, and a mathematical formula relates the new subscripted variables to the earlier subscripted variables. Path conditions, facts, and obligations are accumulated and organized into VCs according to the proof rules in [17]. For example, if a variable v is not changed by a statement s and the subscript before s is i, then the facts known after the statement include $v_{i+1} = v_i$. Control statements and loops introduce implications. For each of the possible control paths (e.g., entering or skipping a loop body), we generate a separate VC. This simplifies the task for the prover, as it explicitly does the requisite case analysis. We have found empirically that this format reduces the complexity of the VCs and their proofs, and increases the chance that Isabelle can prove the VCs automatically.

```
..
lemma 4:

"[|
    ~<(x_4::'obj)> o (q_4::'obj string) = empty_string ;
    reverse(empty_string) o (q_0::'obj string)
      = reverse((s_2::'obj string)) o <x_4> o q_4 ;
    (length ((<x_4> o q_4))) > 0 ;
    is_initial((x_3::'obj)) ;
    is_initial((x_5::'obj))
 |]
 ==>
    reverse(empty_string) o q_0
     = reverse((<x_4> o s_2)) o q_4"

apply (((simp only: simp_thms),clarify?)+)?

apply (force+)?

done
...
end
```

Figure 10: A key VC for the verification of the iterative implementation of Reverse

One VC that needs to be proved for the recursive implementation of `Reverse` is shown in the raw Isabelle output in Fig. 9[2] and in a more human readable format in Fig. 8. This VC comes from the `ensures` clause of the recursive call to `Reverse` in Fig. 7, the satisfaction of which is the essence of the correctness of the implementation. The VC is part of an Isabelle theory file that includes all of the VCs. The `apply(...)` lines are instructions to Isabelle on how to prove each VC. The first section directs Isabelle to perform basic simplifications, such as propositional simplifications. The second line, `apply (force+)?` directs Isabelle to perform the automated reasoning methods. The `done` line indicates to Isabelle that the person thinks the proof is finished. These lines are all generated *automatically* by the VC generator.

Lemma #4 (state index: 5, loop invariant)

$$
\begin{array}{ll}
 & \langle x_4 \rangle * q_4 \neq \Lambda \\
\wedge & \text{reverse}(\Lambda) * q_0 = \text{reverse}(s_2) * \langle x_4 \rangle * q_4 \\
\wedge & |\langle x_4 \rangle * q_4| > 0 \\
\wedge & \text{is\_initial}(x_3) \\
\wedge & \text{is\_initial}(x_5) \\
\Rightarrow & \text{reverse}(\Lambda) * q_0 = \text{reverse}(\langle x_4 \rangle * s_2) * q_4
\end{array}
$$

Figure 11: Human readable version of Fig. 10

[2]The Isabelle versions of the VCs use the `o` symbol for the $*$ concatenation symbol in Resolve's string theory.

	Generation	Proofs
Recursive Time (sec)	0.9	.26
Iterative Time (sec)	1.8	1.37

Table 1: VC Generation and Proof Running Time

In the iterative implementation of `Reverse`, we use a stack to reverse the queue using two loops. Figure 10 is a VC from the loop invariant for the first loop expressed in the Isabelle format; figure 11 expresses the VC in a human readable format. The main string theory lemmas involved here are the associativity of concatenation and the property of concatenation within the `reverse` function.

Isabelle, with the help of the introduced string theory lemmas from Fig. 2, proves both sets of verification conditions automatically. The generation of the VCs and the proofs of the VCs in Isabelle each takes very little time, as seen in Table 1. These timings do not take into account the time for Isabelle to read the Resolve String theory file; the use of Isabelle's internal tools allow for String theory to be incorporated into an Isabelle executable and bypass the time required for Isabelle to process String theory.

Both implementations of the queue `Reverse` code, string theory in Isabelle, and all VCs are on the web at `http://www.cse.ohio-state.edu/~kirschen/rsrg/Isabelle.html`.

4. LESSONS LEARNED

We have shown that the VCs generated for the reverse extension to a queue ADT are automatically provable by an interactive proof assistant without human advice. While these initial results are positive, there are of course many more issues to address. We now describe several of the issues that have come up as we have explored these and other examples. In this section, we use the term "prover" to mean any tool that attempts to prove VCs without the use of specialized decision procedures.

The first potential complication is the addition of quantifiers in `requires` and `ensures` clauses. For a proof of a universally quantified statement in the conclusion of a VC (e.g., $A \Rightarrow \forall x.P(x)$), a prover can simply use a fixed (but arbitrary) element of the universe of the quantification for x, and prove the statement true for that element.

However, when an assumption in a VC involves universal quantification, the natural question is "what term should be used to instantiate the quantified variables?" With this issue, a general proof approach (without the use of a specialized decision procedure) must either never need to instantiate a quantified variable (avoiding the issue), or use a method that instantiates the quantifier "correctly" in many cases (possibly tuned for the types of VCs that are likely to occur). The dual of this is the use of existential quantification. An existential quantification in the assumptions does not cause any problems, whereas an existential quantification in the goal does. Consequently, it is not desirable to have existential quantification in the goal of a VC, although it remains to be seen how often this may arise. One specification design approach is to demand that all requires and ensures clauses and loop invariants be quantifier-free; quantifiers would be introduced only in mathematical definitions. This raises the issue of how to include new definitions in existing theories. One approach is to prove algebraic properties (lemmas) involving those definitions. Using those properties, the automated prover would then attempt to verify the VCs. Another approach is to instead unfold the definition immediately and then let the prover verify the VCs using the expanded version of the definition (exposing any quantifiers in the definition to the prover). While the second approach seems easier to achieve at first glance because the requisite algebraic properties need not be identified and proved, the first approach may have benefits by limiting the complexity of the VCs that the prover works with.

For example, one might want to add the definition `IsPermutation(a,b)` to denote that the string `a` is a permutation of the string `b`. `IsPermutation` may be defined via the number of occurrences of an item in a string. `IsPermutation(a * b, b * a)` is a lemma that should—and can—be proved once and then used to prove many VCs. Fox example, VCs generated for a selection sort algorithm essentially require the prover to deduce `IsPermutation(q1 * (a * <x>),q)` from the assumptions `IsPermutation(q2 * a,q)` and `IsPermutation(q1 * <x>,q2)`. A standard proof involves using lemmas about substitutions, the symmetry of `IsPermutation`, and commutativity of concatenation within the arguments of `IsPermutation`.

5. RELATED WORK

Zee *et al.* [30] have used a hybrid approach of applying both specialized decision procedures and a general proof assistant to prove that code purporting to implement certain data structure specifications is correct. However, the use of Java as a starting language requires that the list specifications use *reference* equality or comparison. Our approach proves properties that depend on the *values* of the objects instead.

Zhang *et al.* [31] describe a decision procedure for queues. Instead of using a special-purpose decision procedure, we use a general-purpose automated proof assistant. The general string theory used for our specifications is slightly simpler than the queue theory developed with the decision procedure, and it is also used to specify the stack ADT used in one of our examples, as well as other Resolve components.

The Why methodology [13] involves a simplified programming language, annotated with logical definitions, axioms, preconditions, post conditions and loop invariants, for which VCs can be generated. A subset of both C (with annotations) and Java (with JML specifications) can be translated into the simplified programming language, such that the VCs generated are claimed to represent the correctness of the original C or Java code. The translation process from C or Java must explicitly capture the memory model of the original source language (C or Java); as a result of using Resolve, we do not need an explicit memory model, simplifying the generated VCs.

SMT solvers such as Yices [11] and Z3 [10] are designed to search for a possible satisfying assignment to a first-order formula by using a SAT solving algorithm (such as DPLL [9, 8]) to find possible satisfying assignments, confirming those assignments via first-order theory-specific satisfiability procedures. SMT solvers are known not to perform well with

quantifiers and the reliance on strictly first-order logic ensures that some predicates may not be definable [19]. We have not yet investigate the relative efficacy of Isabelle compared to any of these SMT solvers.

The Resolve approach for the specifications of a queue ADT is similar to what might be done in JML [20]. For example, the JML specifications for lists use mathematical sequences, similar to our use of strings as the mathematical model for queues and stacks. But with Java, again, the reference/value distinction introduces considerable added complexity.

Resolve superficially resembles the Larch [15] specification and verification discipline. Both approaches employ a programming-by-contract paradigm. Resolve and Larch differ in that Resolve was designed to be used with exactly one programming language and one mathematical specification language, while Larch uses the Larch Shared Language to express mathematical theories (traits) and the Larch Interface Languages (LIL) to express specifications for a particular programming language such as C. Resolve does not have this distinction; the same mathematical language is used both for the creation of mathematical theories and the specification of programmatic operations. Resolve was also designed with the idea of verifiability in mind, so the programming language with its specification language must have a common semantics that allow for proof of soundness and relative completeness of a proof system. Larch was meant to work with several languages whose semantics may be completely different and, indeed, not yet formalized. Also, a key idea of Resolve is that it reuses mathematical theory units as much as possible, while the Larch approach instead tends to reuse a particular trait defined in the Larch Shared Language in each of the interface languages. However, the examples provided for Larch do not show the traits themselves reused within one LIL. For example, a queue and a stack each have a different trait in Larch, while both are modeled by mathematical strings from String theory in Resolve.

6. CONCLUSION AND FUTURE WORK

We have described automated verification of two different implementations of an extension of an ADT. We have also discussed lessons learned about possible challenges to achieving verified software by using and automated proof assistant to prove VCs.

In the future, we expect to create implementations that exercise different parts of the string theory development. For example, the use of substring or permutation definitions in the specifications would require the addition of more string lemmas to continue the automated reasoning process. Finally, we expect to add other theories developed for Resolve specifications into Isabelle, such as finite sets and trees.

Acknowledgments

The authors thank Jeremy Avigad, Harvey M. Friedman, Wayne Heym, Brandon Mintern, Bill Ogden, Murali Sitaraman, and Anna Wolf for their assistance. This work was supported in part by the National Science Foundation under grant DMS-0701260. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] The Coq Proof Assistant Reference Manual Version v8.1.

[2] E. Alkassar and M. A. Hillebrand. Formal functional verification of device drivers. In Shankar and Woodcock [26], pages 225–239.

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

[4] P. Bucci, J. E. Hollingsworth, J. Krone, and B. W. Weide. Part III: implementing components in RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):40–51, 1994.

[5] P. Chalin, P. R. James, and G. Karabotsos. JML4: Towards an industrial grade IVE for java and next generation research platform for JML. In Shankar and Woodcock [26], pages 70–83.

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

[7] M. Daum, J. Dörrenbächer, M. Schmidt, and B. Wolff. A verification approach for system-level concurrent programs. In Shankar and Woodcock [26], pages 161–176.

[8] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[9] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[10] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[11] B. Dutertre and L. de Moura. The Yices SMT solver, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.

[12] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part II: specifying components in RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.

[13] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590/2007 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer-Verlag.

[14] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[15] J. V. Guttag, J. J. Horning, S. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1993.

[16] D. Harms and B. Weide. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[17] W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH, December 1995.

[18] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[19] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–182, New York, NY, USA, 2008. ACM.

[20] G. Leavens. JML language. `http://www.eecs.ucf.edu/~leavens/JML-release/javadocs`.

[21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[22] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: the RESOLVE framework and discipline: a research synopsis. *SIGSOFT Softw. Eng. Notes*, 19(4):23–28, 1994.

[23] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, oct 1998. Springer-Verlag.

[24] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[25] L. Paulson. Set theory for verification: from foundations to functions. *Journal of Automatic Reasoning*, 11(2):353–389, 1993.

[26] N. Shankar and J. Woodcock, editors. *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*. Springer, 2008.

[27] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *ICSR-6: Proceedings of the 6th International Conerence on Software Reuse*, pages 266–283, London, UK, 2000. Springer-Verlag.

[28] A. Starostin and A. Tsyban. Verified process-context switch for c-programmed kernels. In Shankar and Woodcock [26], pages 240–254.

[29] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental Benchmarks for Software Verification Tools and Techniques. In *Proceedings of VSTTE 2008 (Verified Software: Theories , Tools, and Experiments)*. Springer-Verlag, 2008.

[30] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.

[31] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. *Inf. Comput.*, 204(10):1526–1574, October 2006.

SAVCBS 2008 CHALLENGE PROBLEM SOLUTIONS

Formalizing Design Patterns: A Comprehensive Contract for Composite

Jason O. Hallstrom
School of Computing
Clemson University
Clemson, SC 29634-0974
jasonoh@cs.clemson.edu

Neelam Soundarajan
Computer Science and Engineering
Ohio State University
Columbus, OH 43210-1277
neelam@cse.ohio-state.edu

ABSTRACT

Software patterns are used almost universally across design communities as the preferred mechanism for communicating best practice. And while the design archetypes captured by patterns continue to exert significant influence on software design decisions, there is no rigorous foundation for ensuring implementation correctness or reasoning about the systems in which patterns are applied. In this paper, we attempt to identify the conceptual elements necessary of any pattern formalism that satisfies these validation and reasoning objectives. We then present an overview of a particular pattern formalism developed as part of our prior work. The *Composite* pattern is used as a demonstrative example.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages, Methodologies*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Programming by contract, Reliability, Validation*

General Terms

Design, Documentation, Languages, Reliability, Verification

Keywords

Design patterns, pattern contracts, Composite pattern

1. INTRODUCTION

Design patterns began to gain adoption as a mechanism for disseminating best practice after the publication of the seminal *"Gang of Four"* (*GoF*) text [6]. Myriad pattern documentation efforts followed, resulting in a wide range of *pattern catalogs*. Representative efforts include the *"POSA"* series [2,3,9,12], and more specialized efforts devoted to particular implementation technologies (e.g., J2EE, .NET) [1,10].

While each of the patterns contained in these catalogs may not be universally accepted as best practice, one point seems beyond debate: After over a decade of use, patterns continue to exert a significant influence on the design of software, from standard desktop applications to embedded realtime systems and sensor networks.

There is little structural variation among pattern catalogs. Each adopts a variation of the stylized narrative format popularized by the GoF [6]. In this format, a pattern description consists of (*i*) a name, (*ii*) a problem (or objective), (*iii*) structural requirements expressed using UML (or UML-like) notations, (*iv*) code examples, and (*v*) supporting discussion elements. The last component may include a discussion of problem context, implementation pitfalls, system properties arising from a pattern's application, or other issues. And while there is no doubt that this documentation format has proven useful to practitioners, it is also inherently imprecise; patterns lack the foundation necessary to support rigorous validation and reasoning activities. Given the tremendous influence of patterns on software practice and the expectation that this influence will continue in the years ahead, software designers, implementers, and validators must have *precise* pattern specifications — specifications that enable them to reason rigorously about patterns and the systems in which they are applied.

In this paper, we present three contributions. (**C1**) First, we discuss the requisite features of a comprehensive specification formalism for software patterns. We do so by identifying the types of requirements that patterns impose and the dimensions of flexibility that must be preserved in documenting them. Flexibility is, after all, patterns' hallmark — a key contributor to their success and a focal point of our discussion. (**C2**) Second, we present an overview of a *pattern contract* formalism developed as part of our prior work [7,13]. The formalism supports specifications that are both precise and flexible and provides facilities for *pattern specialization*. These specialization facilities enable designers to capture commonly used pattern variants and to arrive at application-specific properties based on the patterns used in a given design. (**C3**) Finally, we apply the formalism to the *Composite* pattern [6][1,2]. Key benefits and limitations are discussed.

Paper Organization. Section 2 describes the require-

[1]Due to space constraints, we assume prior knowledge of the Composite pattern throughout the manuscript.
[2]The discussion is limited to sequential systems in the absence of object aliasing.

ments of a comprehensive pattern specification formalism. Section 3 summarizes our prior work on design pattern contracts. Section 4 presents the Composite pattern contract and discusses an associated specialization. Section 5 highlights elements of closely related work. Section 6 concludes with a discussion of limitations.

2. REQUIREMENTS ON PATTERN FORMALIZATION

On the one hand, it is clear that design patterns impose specific requirements on the classes that play their constituent *roles*. In the case of Composite, for example, it is clear that component, leaf, and composite objects are required to share a set of common interface elements and that each composite is responsible for dispatching calls to its children. On the other hand, it is also understood that patterns are intended to serve as reusable templates; they can be specialized as appropriate for particular scenarios. It is, for example, understood that the operations shared among participants in an instance of Composite will vary, as will the set of calls dispatched to the children of a composite object. This gets to the heart of the problem: An effective pattern formalism must balance the tension between descriptive precision and pattern flexibility. Here we identify the types of requirements imposed by patterns and the dimensions of flexibility that must be preserved.

Structural Requirements. Patterns impose structural requirements on participating objects. These include the roles that may participate in a pattern instance, the signatures that must be provided by objects playing these roles, and the inheritance and association relations among them. The classes that play the roles required by a given pattern will of course vary from one application to another, as will the method signatures they provide to satisfy their role responsibilities. The Leaf role, for instance, will be played by different classes in different applications of the pattern, and the signature of operation() will be implemented in an application-specific manner. Further, each class may provide *multiple* methods intended to play the part of operation(). Or more generally, multiple class methods may correspond to a single role method.

State Requirements. Patterns impose abstract state requirements on participating objects. Objects playing the Composite role, for example, must maintain a set of component objects (as children). It is understood, however, that this set may be implemented using any suitable realization.

Behavioral Requirements – State. Patterns impose behavioral requirements, expressed in terms of standard state-based pre-conditions and post-conditions. The addChild(c) method of Composite, for instance, requires that the component passed as argument not be a member of the composite's child set and ensures that it is added to this set upon termination. As is standard, these requirements can be satisfied in any manner the designer chooses.

Behavioral Requirements – Call Sequence. Patterns not only impose requirements on the state conditions that must be satisfied by particular methods, but also on *how* these conditions must be satisfied. These requirements are expressed in terms of *call sequence conditions* that must be respected during a method's execution. When operation() is invoked on a composite object, for example, it is generally required to place a similar call to its children. Another approach would be for the composite to traverse the tree structure (using getChild()) and invoke appropriate methods on each object that affect the same state changes. While the result would be identical, the implementation would violate a key pattern requirement.

Non-Interference Requirements. Finally, patterns impose implicit requirements on all *non-role* methods provided by participating classes. After all, a pattern describes a slice through a system; participating classes will generally provide method behaviors (and state elements) beyond those required to satisfy their role responsibilities. It is assumed that these behaviors will not interfere with pattern behaviors. A class playing the role of Composite, for example, might include additional (non-role) methods for interacting with the composite's children. These methods must not modify the child set or the intended behavior of the pattern will be compromised.

3. AN OVERVIEW OF A PATTERN CONTRACT FORMALISM

We now consider a pattern formalism designed to provide descriptive precision and pattern flexibility along the identified dimensions. We provide only a brief overview, referring the reader to [7, 13] for a more complete treatment.

In our approach, a pattern is represented by a *contract* that captures the requirements associated with using the pattern correctly and the behavioral guarantees that accrue as a result. *Specializations* of the pattern are represented by a *subcontract*. A subcontract refines a pattern contract to document the manner in which the associated pattern is tailored for use in a given system or to document a *sub-pattern* corresponding to a common usage of the pattern. In this way, contracts capture properties common to all applications of a pattern, while subcontracts capture properties specific to particular applications and sub-patterns.

3.1 Pattern Contracts

A contract consists of four main elements: *role contracts*, a *pattern invariant*, *state abstraction concepts*, and *interaction abstraction concepts*[3]. We describe each of these elements in the remainder of the subsection.

Role Contracts. The contract for a given pattern defines a role contract corresponding to each of the pattern's constituent roles. These specification entities form the core of a pattern specification. Each role contract specifies the abstract state elements, method behaviors, and non-interference conditions that must be satisfied by objects playing the associated roles[4]. The structure mirrors a standard interface specification: Each role contract specifies state elements, method signatures, and corresponding pre- and post-conditions. An additional set of post-conditions may be included to capture non-intereference conditions. These other conditions must be satisfied by all class methods that do not map to one of the role methods.

To specify call sequence requirements, we associate a *ghost variable*, τ (for *"trace"*), with each method invocation. Consider the invocation of a method $m()$. The instance of τ associated with this invocation records information about

[3]A pattern contract may additionally define pattern *instantiation* and *destruction* conditions. We omit these details.
[4]A role contract may additionally define role *enrollment* and *disenrollment* conditions. We omit these details.

the calls placed by $m()$ during its execution. More precisely, τ is an ordered sequence, with each entry corresponding to a single call. The entry records (*i*) the target object, (*ii*) the method invoked, and (*iii*) any argument values passed[5]. Call sequence requirements are then captured as conditions on τ, included as part of $m()$'s post-condition.

Pattern Invariant. A benefit of using many patterns is the behavioral guarantees they afford. Surprisingly, this is true even of *non-behavioral* patterns such as *Composite*, classified as a *structural pattern* by the GoF. As an example, in a standard application of the pattern, certain state conditions can be expected to hold across the nodes within a subtree based on the fact that calls are forwarded to child components. These guarantees are captured by the pattern contract in the form of a *pattern invariant* — a relation on the states of participating objects that holds at well-defined points in the system's execution[6].

Abstraction Concepts. While an application of Composite can be used to ensure that a particular state relation holds within a subtree, the contract cannot define the relation since the definition will vary from one application to another. Similarly, it would be overly-restrictive for the contract to specify the children to which an invocation of operation() must be forwarded since this, too, will vary. To provide this type of flexibility without sacrificing precision, pattern contracts declare *state abstraction concepts* and *interaction abstraction concepts*. The former is a relation on the states of participating objects, used in specifying the pattern invariant and the pre- and post-conditions included within the constituent role contracts. The latter is a relation on instances of τ, used in the specification of call sequence conditions. The key to the flexibility that these concepts provide is that while a pattern contract *declares* the concepts and imposes constraints on the allowable definitions, it defers the *definitions* of the concepts to the subcontracts associated with particular systems and sub-patterns.

3.2 Pattern Subcontracts

The purpose of a subcontract is to specialize a pattern contract so that the resulting specification captures a more specific version of the associated pattern. As such, a subcontract consists of specification elements used to document structural and behavioral refinements to a parent contract.

The first of these specification elements is a *role map*, used in two ways: The most common use is to document the mapping between a role specified in a pattern contract and an application class that plays the role in an application of the pattern. Or stated another way, a role map is used to specify the manner in which a class can be viewed as an instance of its role type. Alternatively, a role map may be used to document a mapping between two roles. In this case, the mapping captures the relationship between a general role and a more specialized version of that role used in a sub-pattern. In each case, a role map consists of a set of *state maps* and *method maps*. The former elements are used to document the realization of the state elements required by a role; they are analogous to abstraction functions. The latter elements are similar, but used to document method realizations. This includes documenting the mappings between signature elements, and argument and return values.

[5]In general, a more sophisticated trace mechanism is required to handle complex call sequence scenarios. We omit consideration of such scenarios.

[6]This relation always holds when control is outside of the objects participating in a pattern instance. While a much stronger guarantee is possible, space limitations preclude its consideration.

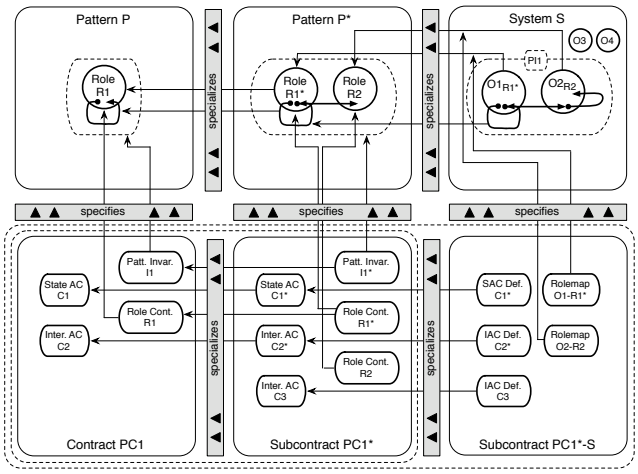


Figure 1: Contracts, Subcontracts, Specializations

Finally, a subcontract specifies refinements to the abstraction concepts specified by the parent contract. These refinements may consist of concept definitions corresponding to a given application, or constraints on the allowable definitions. The latter are used to limit the definitions that my supplied to satisfy the requirements associated with a sub-pattern.

The relationships between patterns, sub-patterns, contracts, subcontracts, and applications are illustrated in Figure 1. In the figure, contract PC1 specifies pattern P. The role contract for R1 specifies implementation requirements on the role, and the pattern invariant I1 specifies an invariant across participating objects. Both are expressed in terms of the state abstraction concept C1 and the interaction abstraction concept C2. The sub-pattern P* specializes pattern P, as documented by subcontract PC1*. Note that this subcontract refines C1, C2, I1, and R1, and additionally adds a new role and a new interaction abstraction concept. Finally, the instance of sub-pattern P* in system S, PI1, is specified by subcontract PC1*-S. The subcontract provides definitions for C1*, C2*, and C3. It additionally provides rolemaps corresponding to objects O1 and O2, which play the roles R1 and R2, respectively.

4. COMPOSITE CONTRACT

We now apply the formalism to a common variant of the Composite pattern. For the sake of presentation, the contract has been segmented into separate listings. We describe each segment in turn.

The contract begins by declaring the state abstraction concepts used throughout the remainder of the document (Listing 1). The first, Modified(), captures the notion of a *significant change* to a composite object with respect to one of its children. At the point this concept is used, it determines the set of children that must receive a forwarded operation() call from a composite. More precisely, given the pre-conditional and post-conditional states of the target composite and one of its children, Modified() determines whether a call to operation() must be forwarded to the child.

```
1  pattern contract Composite {
2
3   state abstraction concepts:
4    Modified(Composite_α, Composite_β, Component_γ)
5    Consistent(Component_δ, Component_ε)
6    constraints:
7     (↑ α =↑ β) ∧ ¬((↑ δ =Leaf) ∧ (↑ ε =Leaf))∧
8      ∀c1, c1* ⊢ Composite, c2 ⊢ Component ::
9       ((Consistent(c1, c2) ∧ ¬Modified(c1, c1*, c2))
10        ⟹ Consistent(c1*, c2))
11
12  interaction abstraction concepts:
13   ...omitted...
14
15  pattern invariant:
16   ∀c1, c2 ⊢ Component :
17    (c1 ∈players) ∧ (c2 ∈players)∧
18    (↑↑ c1 =Component) ∧ (c2 ∈ c1.children)) :
19     ((c2.parent= c1)∧Consistent(c1, c2))
```

Listing 1: Composite Contract (part 1)

The second concept, Consistent(), is used to capture the notion of *state consistency* between a composite and a child. It is used in the post-condition of operation() to require that the method leave the target object in a state that is *consistent* with its parent. As we will see, it will also be used in expressing the pattern invariant.

The constraints clause restricts the concept definitions that may be supplied in a subcontract to ensure that the pattern invariant is satisfied. Three restrictions are imposed. First, the constraints require that the first two arguments of Modified() be of the same type (since this operation is only applied on two states of the same object in the contract). The "↑" notation denotes the application class (or specialized role) mapped to the target's type. Second, at least one of the arguments to Consistent must not be a leaf (since this concept captures consistency between a parent and a child — a relationship that cannot hold between two leafs.) Finally, the last conjunct requires that if two states of a composite are considered to be sufficiently similar according to Modified(), and the first is *consistent* with a given child, so too, must the second. This is necessary since the definition of Modified() controls whether operation() calls are forwarded — calls which are in turn responsible for ensuring *consistency* between parents and children.

For the sake of presentation, we provide a simplified contract, omitting interaction abstraction concepts.

Finally, the contract specifies the pattern invariant. If all implementation requirements are satisfied, Composite ensures that every child component is *consistent* —according to an appropriate definition— with its parent component.

Next, the contract specifies the role contract for the Component role (Listing 2). The notational elements within brackets indicate that exactly one class must be mapped to this role in an application of the pattern, and this class must be abstract.

The body of the role contract begins by requiring that classes playing the role maintain a Component reference, referred to as parent in the specification. As the name suggests, this variable is intended to store a reference to the component's parent, if any, in the composite tree[7].

[7]In general, it is more flexible to treat parent as a ghost variable, providing developers the ability to omit its realization.

```
1  role contract Component [1,abstract] {
2
3   Component parent;
4
5   void operation();
6     pre: true
7    post: (parent= #parent)∧
8          Consistent(parent, this)
9
10  others:
11   post: (parent= #parent)∧
12         (Consistent(parent, #this)
13            ⟹ (Consistent(parent, this))
14 }
```

Listing 2: Composite Contract (part 2)

Next, the role contract provides the specification of operation(), and an others clause used to capture the conditions that must be satisfied by all *non-role* methods supplied by classes playing the role. The specification of operation() requires that the method preserve the parent reference and leave the target object in a state that is *consistent* with its parent. The non-intereference conditions are identical, but the consistency requirement is only imposed if the target was in a consistent state prior to the call to operation().

```
1  role contract Composite [+] : Component {
2
3   Set<Component> children;
4
5   void add(Component c);
6     pre: c ∉ children
7    post: (children= (#children∪{c}))∧
8          (c.parent=this)∧
9           ∀oc ⊢ Component :
10            (oc ∈ #children) :
11             ¬Modified(this, #this, oc)∧
12         (|τ.c.operation| = 1)
13
14  void remove(Component c);
15    pre: c ∈ children
16   post: (children= (#children−{c}))∧
17         ∀oc ⊢ Component :
18          (oc ∈ #children) :
19           ¬Modified(this, #this, oc)
20
21  ...other child management methods omited...
22
23  void operation();
24    pre: ...inherited from Component...
25   post: ...inherited from Component...∧
26         (children= #children)∧
27         ∀c ⊢ Component :
28          (c ∈children) :
29            (Modified(this, #this, c)
30              ⟹ (|τ.c.operation| = 1))
31
32  others:
33    ...inherited from Component...∧
34    (children=children)∧
35    ∀c ⊢ Component :
36     (c ∈ #children) :
37      ¬Modified(this, #this, c)
38 }
```

Listing 3: Composite Contract (part 3)

```
1 role contract Leaf [*] : Component {
2
3  void operation();
4    ...inherited from Component...
5
6  others:
7    ...inherited from Component...
8 }
```

Listing 4: Composite Contract (part 4)

The bulk of the contract is devoted to specifying the Composite role (Listing 3). The first line of the role contract indicates that one or more classes must be mapped to this role in an application of the pattern, and each must inherit from the class mapped to the Component role.

As before, the contract begins with state requirements: Participating classes must maintain a Set of component objects. This variable, children, stores references to each of the composite's children.

Next, the contract specifies the method behaviors required of composite objects: First, participating classes must supply child management methods. The pre-condition of add(), for example, requires that the child passed as argument not be contained within children. The method is required to add the child to children and assign itself as the child's parent. The next conjunct requires that the composite not be significantly modified (according to Modified()) by the call.

More interesting is the last conjunct, which specifies a call sequence requirement: $|\tau.\mathsf{c.operation}|$ denotes the subsequence obtained by projecting τ on object c and method operation(). Hence, the clause requires that the composite invoke operation() on the new child. While this requirement is not discussed in the original pattern description, it is essential to ensuring the pattern invariant; without it, there is no guarantee that the child will be in a state consistent with its parent. Requirements on remove() are analogous, but omit call sequence requirements. Other management methods have been elided.

The pre-condition on operation() is inherited from Component; it is trivially *true*. The inherited post-condition is strengthened: first, it requires that the children variable not be altered. More interestingly, it requires that if operation() modify the state of the component in a manner that is significant with respect to some child, the object is responsible for invoking operation() on that child. This ensures that if the original call breaks the pattern invariant, the forwarding behavior will re-assert the invariant.

The non-interference conditions specified in the others clause strengthen the conditions specified by the Component role contract. In particular, non-role methods of a class mapped to Composite are required to preserve the children variable. Further, they are not allowed to modify the state of the composite in a *significant* way.

Finally, the contract specifies the role contract for Leaf (Listing 4). The declaration indicates that zero or more classes may map to this role and each must inherit from the class mapped to Component. The remainder of the role contract is inherited without change.

To arrive at the implementation requirements and behavioral guarantees associated with a particular application of Composite, a corresponding subcontract must be specified. It is the composition of the subcontract and the contract that guides system implementation activities and assists in reasoning about *pattern-centric* behaviors. As an example, consider a standard application of the pattern in the context of designing a GUI library. Classes within the library might represent windows, frames, panels, and other graphical elements, and the tree structure imposed by Composite would mirror visual containment relationships. The subcontract for this application would provide role maps for each of the participating classes; the details are straightforward. More interesting are the concept definitions.

For simplicity, we assume that only one method plays the role of operation() — namely, a resize() method used to adjust the size of a visual container and all of its children. In this scenario, the definition of Modified() would rely only on the first two arguments: The relation would evaluate to *true* if the object states passed as argument had different width and height values, and *false* otherwise. Similarly, the definition of Consistent() would evaluate to *true* if the component states passed as argument had equal dimensions, and *false* otherwise. By substituting these definitions into the role contracts, application-specific requirements emerge. And by satisfying these requirements, developers are assured of the specialized pattern invariant: When control is outside of the participating objects, the dimensions of the children in any subtree total the dimensions of the parent. In this way, the contract formalism captures precise implementation requirements while affording flexible specialization to document applications and sub-patterns[8].

5. RELATED WORK

The benefits and pitfalls of pattern formalization have been discussed by other authors. A number of specification formalisms have been proposed. Here we briefly survey four representative efforts.

Eden and Hirshfeld [5] focus on specifying the *structural* (i.e., static) properties of design patterns. The authors describe a higher-order logic formalism in which patterns are specified as formulae. The basic terms of the logic consist of classes and methods. The associated relations correspond to standard syntactic concepts, including class membership, method invocation, and inheritance. Each pattern is specified as a list of participants (i.e., classes and methods) and the relations among them. While the approach handles rich structural properties, it does not provide facilities for state abstraction, pattern specialization, or behavioral properties.

In contrast to Eden and Hirshfeld's structural emphasis, Mikkonen [11] focuses on *behavioral* (i.e., dynamic) properties. Using his approach, patterns are expressed in an action system notation with roots in the UNITY [4] formalism for parallel and distributed systems. Each pattern is specified as a set of state elements, relations on these elements, and guarded assignments. Refinement is supported through *superposition*; specification layers can be composed without violating safety properties as long as each layer writes only to the state components it defines. While this approach offers a number of interesting benefits, including the ability to concisely specify complex temporal properties, it is, in a sense, *too abstract*. Structural and control-flow requirements are intentionally abstracted away, compromising the

[8]Sub-patterns are documented in an analogous matter; we omit there consideration.

efficacy of the formalism as a prescriptive mechanism for software design.

Taibi and Ngo [14] describe a hybrid approach. Their formalism relies on predicate logic to capture structural properties of patterns, and an action system notation to capture behavioral properties. In effect, their work combines the key elements proposed by Eden and Hirshfeld, and Mikkonen.

Interestingly, one of the most comprehensive notations for specifying patterns predates the GoF's text. Helm et al. [8] describe a notation for specifying *"behavioral compositions"* in object-oriented software, including those captured by patterns. There are a number of similarities with our work: A pattern is represented as a contract that specifies the participating roles, their required state elements and method behaviors, a pattern invariant, and instantiation conditions. The notation also provides facilities for contract composition and refinement and documenting the mappings between application classes and contract participants. While the underlying concepts seem essential, the realization lacks both precision and flexibility. Fundamental precision limitations include an inability to document method pre-conditions and a weakly expressive notation for documenting call sequence conditions. It is not, for example, possible to restrict the invocations made between calls or to relate the arguments and return values of successive calls. Fundamental flexibility limitations include the absence of state abstraction and the use of name-based (i.e., syntactic) mappings between application classes and contract participants.

6. CONCLUSION

We conclude by noting that the contract formalism presented here is an abbreviated version of a more complete specification notation. The same is true of the pattern contract for Composite. In particular, the trace mechanism has been simplified to accommodate space requirements, with important consequences on both the precision and flexibility of the resulting contract.

Consider, for example, the specification of operation() in Listing 3. The post-condition requires that if the method alters the state of the composite in a *significant* way (i.e., according to the definition of Modified()), the method must in turn invoke operation() on each affected child to re-assert the pattern invariant. But what if, after forwarding the appropriate calls, the method again modifies the state of the composite? Or alternatively, what if a call from the method re-enters the Component hierarchy above the executing node and modifies component state? And what happens if control is *always* within a participating object? When does the pattern invariant hold?

Addressing these questions hinges on the use of a more sophisticated trace mechanism. We have recently developed the concept of a *pattern-instance trace*, a behavioral projection on a single pattern instance, which we believe provides an elegant solution. We are currently experimenting with the notation and hope to discuss early results if invited for presentation at the workshop.

Acknowledgments

This work is supported by the National Science Foundation through awards CNS-0745846 and DUE-0633506.

7. REFERENCES

[1] D. Alur, D. Malks, J. Crupi, G. Booch, and M. Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Mountain View, CA, USA, 2003.

[2] F. Buschmann, K. Henney, and D. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2007.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[4] K. Chandy. *Parallel Program Design: A Foundation*. Addison-Wesley, Boston, MA, USA, 1988.

[5] A. Eden and Y. Hirshfeld. Principles in formal specification of object-oriented design and architecture. In *The 2001 Conference of the Centre for Advanced Studies on Collaborative Research*, pages (*cd–rom*), Indianapolis, IN, USA, November 2001. IBM Press.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[7] J. Hallstrom, N. Soundarajan, and B. Tyler. Amplifying the benefits of design patterns: From specification through implementation. In *Foundational Approaches to Software Engineering*, pages 214–229, Berlin, Germany, March 2006. Springer-Verlag.

[8] R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–180, New York, NY, USA, October 1990. ACM.

[9] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, New York, NY, USA, 2004.

[10] Microsoft Corporation. *Enterprise Solution Patterns Using Microsoft .NET*. Microsoft Press, Redmond, WA, USA, 2003.

[11] T. Mikkonen. Formalizing design patterns. In *The 20^{th} th International Conference on Software Engineering*, pages 115–124, Los Alamitos, CA, USA, April 1998. IEEE Computer Society.

[12] D. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[13] N. Soundarajan and J. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *The 26^{th} International Conference on Software Engineering*, pages 666–675, Los Alamitos, CA, USA, May 2004. IEEE Computer Society.

[14] T. Taibi and D. Ngo. Formal specification of design patterns – a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

Verifying the Composite Pattern using Separation Logic

Bart Jacobs * Jan Smans † Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium

{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract

Often, a module developer wishes to expose a graph of objects to client code, allowing client code to access the graph through any node directly, while maintaining hidden consistency conditions over the graph. In this note, we describe how to specify and verify such code using separation logic, using as an example a binary tree structure where each node keeps a count of its descendant nodes. The idea is to describe the tree structure as the separate conjunction of *the focus node's subtree* and *the focus node's context*. The description can be rewritten to use any other node as the focus node at any time. This enables an elegant modular proof of the tree implementation on the one hand, and client code on the other hand.

We describe how we verified an example program using the VeriFast program verifier prototype.

1. Introduction

Classical ownership systems force all accesses of a composite object structure to go through a designated root object. This makes it difficult to verify programs where a module exposes a graph of objects to client code, and client code accesses the component objects directly. In this paper we show an approach for verifying such programs using separation logic (Reynolds 2002). The general idea is that before accessing a node, the separation logic assertion that describes the structure is rewritten so that the target node is at the "top level" of the recursive description and can easily be separated out. For information hiding, a single abstract predicate (Parkinson and Bierman 2005) is used to represent the entire structure (Parkinson 2007).

2. Example

We illustrate the approach by showing a full functional correctness specification, an implementation, and client code for a tree structure module written in C. The implementation keeps a count in each node of the node's descendant nodes. The code is annotated so that it is verified automatically with our VeriFast tool (Jacobs and Piessens 2008), a program verifier based on symbolic execution with a separation logic representation of memory.

2.1 Specification and client

The specification of the tree module is shown in Figure 1. The module's functions, $create\_tree$, $tree\_add\_left$, $tree\_add\_right$, $tree\_get\_count$, $tree\_get\_parent$, and $tree\_dispose$, are specified in terms of the abstract predicate $tree(n, c, t)$, where n is the *focus node*, c is the focus node's *context*, and t is the focus node's *subtree*. The context and the subtree are values of the inductive datatypes $context$ and $tree$, respectively. The context is the part of the tree obtained by removing the focus node and its subtree; it is either $root$, indicating that the focus node is the root node of the tree; or $left\_context(pc, p, r)$, indicating that the focus node is the left child of a node p, whose own context is pc and whose right child's subtree is r; or $right\_context(pc, p, l)$, analogously indicating that the focus node is the right child of a node p, whose own context is pc and whose left child's subtree is l. The subtree is either nil, indicating that the focus node is a null pointer; or $tree(n, l, r)$, indicating that the focus node is n and its left and right child's subtrees are l and r, respectively.

The specification of function $tree\_get\_count$ uses the pure primitive recursive function $count$ over $tree$ values.

Predicate assertion arguments may be *patterns*, of the form $?x$, which bind x in the remainder of the symbolic execution. They are in effect existentially quantified logical variables.

Throughout, the star operator ($*$) indicates separate conjunction, and binds like conjunction, i.e. more weakly than relational operators.

Function $tree\_add\_left$ not only creates a new node and adds it as the left child of the specified node; it also makes the new node the new focus node. Similarly, $tree\_add\_right$ and $tree\_get\_parent$ make the new right child node and the parent node the focus node, respectively. As a result, the example client program in Figure 2 verifies as is.

2.2 Implementation and proof

The implementation of the tree module is shown in Figures 3 – 7. Figure 3 shows how predicate $tree$ is defined in terms of predicates $context$ and $subtree$. A predicate assertion $subtree(n, p, t)$ states that the heap contains a representation of a subtree t with root node n and parent pointer p, with consistent $count$ fields. An assertion $context(n, p, count, c)$ states that the heap contains a representation of a context c of node n with parent p, where the $count$ fields are consistent under the assumption that the count of the subtree at n is $count$. The assertion $malloc\_block\_node(n)$ indicates that pointer n points to a **struct** $node$ object allocated using $malloc$.

Figure 4 shows the implementations of $create\_node$ and $create\_tree$. A **close** $p(\overline{v})$; ghost statement consumes the body of predicate p, with arguments $\overline{v}$ substituted for the parameters, removing from the symbolic heap the points-to assertions and abstract predicate assertions mentioned in the body, and then produces, i.e. adds to the symbolic heap, the abstract predicate assertion $p(\overline{v})$ itself.

Figures 5 and 6 show the implementation of functions $tree\_add\_left$ and $tree\_get\_count$, and auxiliary functions $subtree\_get\_count$ and $fixup\_ancestors$. Function $subtree\_get\_count$ relies on the count invariant expressed in the $subtree$ predicate so that it can simply return the value of the $count$ field. Function $tree\_add\_left$ first creates the new node, and then calls $fixup\_ancestors$, which recursively ascends the tree to adjust the $count$ fields of the new node's ancestors. The effect of the $fixup\_ancestors$ calls is framed

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

† Jan Smans is a Research Assistant of the Research Foundation - Flanders (FWO)

```
struct node;
typedef struct node *node;

inductive tree := nil | tree(node, tree, tree);

fixpoint int count(tree t) {
  switch (t) {
    case nil : return 0;
    case tree(n, l, r) : return 1 + count(l) + count(r);
  }
}

inductive context :=
  | root
  | left_context(context, node, tree)
  | right_context(context, node, tree);

predicate tree(node node, context c, tree subtree);

node create_tree();
  requires emp;
  ensures tree(result, root, tree(result, nil, nil));

node tree_add_left(node node);
  requires
    tree(node, ?c, ?t) *
    switch (t) {
      case nil : false;
      case tree(n0, l, r) : l = nil;
    };
  ensures
    switch (t) {
      case nil : false;
      case tree(n0, l, r) :
        tree(result, left_context(c, node, r),
          tree(result, nil, nil));
    };

node tree_add_right(node node);
  ... analogous ...

int tree_get_count(node node);
  requires tree(node, ?c, ?t);
  ensures tree(node, c, t) * result = count(t);

node tree_get_parent(node node);
  requires tree(node, ?c, ?t) * c ≠ root * t ≠ nil;
  ensures
    switch (c) {
      case root : false;
      case left_context(pns, p, r) :
        result = p * tree(p, pns, tree(p, t, r));
      case right_context(pns, p, l) :
        result = p * tree(p, pns, tree(p, l, t));
    };

void tree_dispose(node node);
  requires tree(node, root, _);
  ensures emp;
```

Figure 1. Specification of the tree module. Annotations are shown on a gray background.

```
int main()
  requires emp;
  ensures emp;
{
  // {}
  node node := create_tree();
  // {tree(n0, root, tree(n0, nil, nil))}
  node := tree_add_left(node);
  // {tree(n1, left_context(root, n0, nil), tree(n1, nil, nil))}
  node := tree_add_right(node);
  // {tree(n2, right_context(left_context(root, n0, nil), n1,
  //     nil), tree(n2, nil, nil))}
  node := tree_get_parent(node);
  // {tree(n1, left_context(root, n0, nil),
  //     tree(n1, nil, tree(n2, nil, nil)))}
  node := tree_add_left(node);
  // {tree(n3, left_context(left_context(root, n0, nil), n1,
  //     tree(n2, nil, nil)), tree(n3, nil, nil))}
  node := tree_get_parent(node);
  // {tree(n1, left_context(root, n0, nil), tree(n1,
  //     tree(n3, nil, nil), tree(n2, nil, nil)))}
  node := tree_get_parent(node);
  // {tree(n0, root, tree(n0, tree(n1, tree(n3,
  //     nil, nil), tree(n2, nil, nil)), nil))}
  tree_dispose(node);
  // {}
  return 0;
}
```

Figure 2. An example client program. Symbolic states are shown in blue.

by the fact that this function requires access only to the current node's context, not to the entire tree. A *fixup_ancestors* call takes a context that is consistent with respect to some unknown count, and returns the same context, made consistent with respect to the given count.

Ghost statement **open** $p(\overline{pat})$; is the reverse of **close**: it consumes the abstract predicate assertion $p(\overline{pat})$ and then produces the predicate's body. Contrary to the **close** statement, the arguments in an **open** statement can be patterns; the scope of the variables bound by these patterns extends to the end of the function body.

Function *tree_add_left* contains an **assert** statement; its only purpose here is to bind variable r to the *tree* value that represents the subtree below node *nodeRight*.

Figure 7 shows the implementation of functions *tree_get_parent* and *tree_dispose*, and auxiliary function *subtree_dispose*. As pointed out before, function *tree_get_parent* not only returns the parent node pointer of the specified node, but also makes the parent node the focus node of the description of the tree structure. The implementation effectively moves the specified node's parent's fields, and the specified node's sibling subtree, from the context part of the tree description into the subtree part.

2.3 Non-contiguous focus changes

The tree module specification, as shown in Figure 1, requires client code to navigate the tree contiguously; i.e., to access a given node, the client must navigate to this node using *tree_get_parent*, *tree_get_left*, and *tree_get_right* calls (the latter functions are not shown). In this subsection, we show how the tree module's specification can be extended so that clients can change focus not only to adjacent nodes but to any node in the tree.

```
struct node {
  struct node *left;
  struct node *right;
  struct node *parent;
  int count;
};

predicate subtree(node node, node parent, tree t) =
  switch (t) {
    case nil : node = 0;
    case tree(node0, leftNodes, rightNodes) :
      node = node0 * node ≠ 0 *
      node→left ↦ ?left *
      node→right ↦ ?right *
      node→parent ↦ parent *
      node→count ↦ count(t) *
      malloc_block_node(node) *
      subtree(left, node, leftNodes) *
      subtree(right, node, rightNodes);
  };

predicate context(node n, node p, int count, context c) =
  switch (c) {
    case root : p = 0;
    case left_context(pns, p0, r) :
      p = p0 * p ≠ 0 *
      p→left ↦ n *
      p→right ↦ ?right *
      p→parent ↦ ?gp *
      p→count ↦ ?pcount *
      malloc_block_node(p) *
      context(p, gp, pcount, pns) *
      subtree(right, p, r) *
      pcount = 1 + count + count(r);
    case right_context(pns, p0, l) :
      ... analogous ...
  };

predicate tree(node node, context c, tree subtree) =
  context(node, ?parent, count(subtree), c) *
  subtree(node, parent, subtree);
```

Figure 3. Struct and predicate declarations

The additional specification elements are shown in Figure 8. The main new element is the specification of the *change_focus lemma function*. A lemma function, or *lemma* for short, is like a regular C function, but it is declared in an annotation; it is not allowed to have any side-effects; and the verifier checks that it terminates. Unlike a regular function call, a lemma function call is a ghost statement. The sole purpose and effect of calling a lemma function is to rewrite the symbolic state into a different, but equivalent, form. Lemma *change_focus* takes a tree structure description with focus node $n0$, and a node pointer n that is contained in the tree structure at a path p, and rewrites the tree structure description so that the focus node is n. The new *context* and *tree* values are specified using the fixpoint functions *combine*, *context_at*, and *subtree_at*.

Figure 9 shows an example of a piece of client code whose verification is enabled by lemma *change_focus*.

Figure 10 shows the implementation, i.e. the proof, of lemma *change_focus*. The proof uses two auxiliary lemmas, *go_to_root* and *go_to_descendant*. Lemma *go_to_root* operates by recursion

```
node create_node(node p)
  requires emp;
  ensures subtree(result, p, tree(result, nil, nil));
{
  node n := malloc(sizeof(struct node));
  n→left := 0; close subtree(0, n, nil);
  n→right := 0; close subtree(0, n, nil);
  n→parent := p;
  n→count := 1;
  close subtree(n, p, tree(n, nil, nil));
  return n;
}

node create_tree()
  requires emp;
  ensures tree(result, root, tree(result, nil, nil));
{
  node n := create_node(0);
  close context(n, 0, 1, root);
  close tree(n, root, tree(n, nil, nil));
  return n;
}
```

Figure 4. Implementation of function *create_tree*

(i.e., induction) on the structure of the *context* value; similarly, lemma *go_to_descendant* operates by recursion (induction) on the structure of the *path* value.

VeriFast checks that each lemma function terminates, by disallowing loops and indirect recursive calls, and by checking that at each direct recursive call, either the callee's footprint is a strict subset of the caller's footprint, or the lemma function's body is a switch statement on one of its parameters, and the value of this parameter for the callee is a component of the value of this parameter for the caller. The footprint of a call is the footprint of the precondition; the footprint of an assertion is the set of memory locations that are present in each heap that satisfies the assertion. VeriFast checks the footprint requirement for a given recursive call by checking that after consuming the callee's precondition, at least one points-to assertion is left in the symbolic state.

3. Conclusion

We propose a way to specify and verify modules that expose a graph of objects, while maintaining hidden invariants over this graph and allowing clients to access the graph at any node directly. In this approach, the graph structure is described by a separation logic assertion. Before accessing a node, the assertion is rewritten to separate out this node. We illustrated the approach by showing a specification and an implementation of a binary tree module, annotated to enable verification using our VeriFast program verifier.

VeriFast is available at

http://www.cs.kuleuven.be/~bartj/verifast/

References

Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report 520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

```
int subtree_get_count(node node)
  requires subtree(node, ?parent, ?nodes);
  ensures subtree(node, parent, nodes) *
    result = count(nodes);
{
  int result := 0;
  open subtree(node, parent, nodes);
  if (node ≠ 0) { result := node→count; }
  close subtree(node, parent, nodes);
  return result;
}

void fixup_ancestors(node n, node p, int count)
  requires context(n, p, _, ?c);
  ensures context(n, p, count, c);
{
  open context(n, p, _, c);
  if (p ≠ 0) {
    node left := p→left;
    node right := p→right;
    node grandparent := p→parent;
    int leftCount := 0;
    int rightCount := 0;
    if (n = left) {
      leftCount := count;
      rightCount := subtree_get_count(right);
    } else {
      leftCount := subtree_get_count(left);
      rightCount := count;
    }
    {
      int pcount := 1 + leftCount + rightCount;
      p→count := pcount;
      fixup_ancestors(p, grandparent, pcount);
    }
  }
  close context(n, p, count, c);
}
```

Figure 5. Helper functions for function *tree_add_left*

Matthew Parkinson. Class invariants: the end of the road? In *IWACO 2007*, 2007.

Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL 2005*, 2005.

J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS 2002*, 2002.


```
node tree_add_left(node node)
  requires
    tree(node, ?c, ?t) *
    switch (t) {
      case nil : false;
      case tree(n0, l, r) : l = nil;
    };
  ensures
    switch (t) {
      case nil : false;
      case tree(n0, l, r) :
        tree(result, left_context(c, node, r),
          tree(result, nil, nil));
    };
{
  open tree(node, c, t);
  node n := create_node(node);
  open subtree(node, ?parent, t);
  node nodeRight := node→right;
  assert subtree(nodeRight, node, ?r);
  {
    node nodeLeft := node→left;
    open subtree(nodeLeft, node, nil);
    node→left := n;
    close context(n, node, 0, left_context(c, node, r));
    fixup_ancestors(n, node, 1);
  }
  close tree(n, left_context(c, node, r), tree(n, nil, nil));
  return n;
}

node tree_add_right(node node)
  ... analogous ...

int tree_get_count(node n)
  requires tree(n, ?c, ?t);
  ensures tree(n, c, t) * result = count(t);
{
  open tree(n, c, t);
  int result := subtree_get_count(n);
  close tree(n, c, t);
  return result;
}
```

Figure 6. Implementation of function *tree_add_left*

```
node tree_get_parent(node node)
  requires tree(node, ?c, ?t) ∗ c ≠ root ∗ t ≠ nil;
  ensures
    switch (c) {
      case root : false;
      case left_context(pns, p, r) :
        result = p ∗ tree(p, pns, tree(p, t, r));
      case right_context(pns, p, r) :
        result = p ∗ tree(p, pns, tree(p, l, t));
    };
{
  open tree(node, c, t);
  open subtree(node, _, t);
  node p := node→parent;
  close subtree(node, p, t);
  open context(node, p, count(t), c);
  assert context(p, ?gp, ?pcount, ?pns);
  switch (c) {
    case root :
    case left_context(pns, p0, r) :
      close subtree(p, gp, tree(p, t, r));
    case right_context(pns, p0, l) :
      close subtree(p, gp, tree(p, l, t));
  }
  assert subtree(p, gp, ?pt);
  close tree(p, pns, pt);
  return p;
}

void subtree_dispose(node node)
  requires subtree(node, _, _);
  ensures emp;
{
  open subtree(node, _, _);
  if (node ≠ 0) {
    subtree_dispose(node→left);
    subtree_dispose(node→right);
    free(node);
  }
}

void tree_dispose(node node)
  requires tree(node, root, _);
  ensures emp;
{
  open tree(node, root, _);
  open context(node, _, _, root);
  subtree_dispose(node);
}
```

Figure 7. Implementation of functions *tree_get_parent* and *tree_dispose*

```
fixpoint tree combine(context c, tree t) {
  switch (c) {
    case root : t;
    case left_context(pns, p, right) :
      combine(pns, tree(p, t, right));
    case right_context(pns, p, left) :
      combine(pns, tree(p, left, t));
  }
}

inductive path := here | left(path) | right(path);

fixpoint bool contains_at(tree t, path p, node n) {
  switch (t) {
    case nil : return false;
    case tree(rootNode, l, r) : return
      switch (p) {
        case here : n = rootNode;
        case left(p) : contains_at(l, p, n);
        case right(p) : contains_at(r, p, n);
      };
  }
}

fixpoint context context_at(context c, tree t, path p) {
  switch (p) {
    case here : return c;
    case left(p) : return switch (t) {
      case nil : c;
      case tree(n, l, r) :
        context_at(left_context(c, n, r), l, p);
    };
    case right(p) : ...analogous...
  }
}

fixpoint tree subtree_at(tree t, path p)) {
  switch (t) {
    case nil : return nil;
    case tree(n, l, r) : return switch (p) {
      case here : t;
      case left(p) : subtree_at(l, p);
      case right(p) : subtree_at(r, p);
    };
  }
}

lemma void change_focus(node n0, path p, node n);
  requires tree(n0, ?c, ?t) ∗ contains_at(combine(c, t), p, n);
  ensures tree(n, context_at(root, combine(c, t), p),
    subtree_at(combine(c, t), p));
```

Figure 8. Specification of lemma *change_focus*

```
int main()
  requires emp;
  ensures emp;
{
  node root := create_tree();
  node l := tree_add_left(root);
  node lr := tree_add_right(l);
  change_focus(lr, left(here), l);
  node ll := tree_add_left(l);
  change_focus(ll, left(right(here)), lr);
  node lrr := tree_add_right(lr);
  change_focus(lrr, here, root);
  tree_dispose(root);
  return 0;
}
```

Figure 9. Client code that uses *change_focus* for non-contiguous navigation

```
lemma void go_to_root(node n, context c)
  requires tree(n, c, ?t);
  ensures tree(_, root, combine(c, t));
{
  switch (c) {
    case root :
    case left_context(pcn, p, r) :
      open tree(n, c, t);
      open context(n, _, _, _);
      assert context(p, ?gp, _, _);
      close subtree(p, gp, tree(p, t, r));
      go_to_root(p, pcn);
    case right_context(···) : ... analogous ...
  }
}

lemma void go_to_descendant(node n0, path p, node n)
  requires tree(n0, ?c, ?t) * contains_at(t, p, n);
  ensures tree(n, context_at(c, t, p), subtree_at(t, p));
{
  switch (p) {
    case here :
      open tree(n0, c, t);
      open subtree(n0, ?p, t);
      switch (t) {
        case nil :
        case tree(n00, l, r) :
          close subtree(n0, p, t);
          close tree(n0, c, t);
      }
    case left(p) :
      open tree(n0, c, t);
      open subtree(n0, ?p, t);
      switch (t) {
        case nil :
        case tree(n00, l, r) :
          node left := n0→left;
          close context(left, n0, count(l),
            left_context(c, n0, r));
          close tree(left, left_context(c, n0, r), l);
          go_to_descendant(left, p, n);
      }
    case right(p) : ... analogous ...
  }
}

lemma void change_focus(node n0, path p, node n)
  requires tree(n0, ?c, ?t) * contains_at(combine(c, t), p, n);
  ensures tree(n, context_at(root, combine(c, t), p),
    subtree_at(combine(c, t), p));
{
  go_to_root(c);
  assert tree(?rootNode, _, _);
  go_to_descendant(rootNode, p, n);
}
```

Figure 10. Proof of lemma *change_focus* and auxiliary lemmas

Permissions to Specify the Composite Design Pattern

Kevin Bierhoff Jonathan Aldrich
Institute for Software Research, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{kevin.bierhoff,jonathan.aldrich} @ cs.cmu.edu

ABSTRACT

The Composite design pattern is a well-known implementation of whole-part relationships with trees of Composite objects. This paper presents a permission-based specification of the Composite pattern that allows nodes in an object hierarchy to depend on invariants over their children while permitting clients to add new children to any node in the hierarchy at any time. Permissions can capture the circular dependencies between nodes and their children that arise in this context. The paper also discusses verifying a Composite implementation and known limitations of the presented specification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, languages, verification.

Keywords

Typestate, invariants, implementation verification.

1. INTRODUCTION

The Composite design pattern is a well-known implementation of whole-part relationships with trees of Composite objects [7]. If nodes depend on invariants over their children then it becomes challenging to verify that adding a child to a node correctly notifies the node's parents of changes [9]. In particular, these circular dependencies between nodes are hard to capture with verification approaches based on ownership [1] or uniqueness [5].

This paper presents a permission-based specification of the Composite design pattern that allows nodes in an object hierarchy to depend on invariants over their children while permitting clients to add new children to any node in the hierarchy at any time (section 3). Permissions can capture the circular dependencies between nodes and their children that arise in this context. Section 4 outlines how the presented specification can be used for verifying a simple Composite implementation.

The approach is based on the authors' work on sound reasoning about typestates in object-oriented programs [3] (briefly introduced in section 2). We therefore use a typestate-based invariant in our presentation. Section 5 discusses how this and other limitations of the presented specification can be remedied before section 6 concludes.

7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008), November 9–10, 2008, Atlanta, GA, USA.

2. PERMISSIONS

This section gives a brief introduction to the approach used to specify the Composite pattern in the following section. The approach was originally developed for sound reasoning about typestates in object-oriented programs with aliasing [3] and is based, in part, on previous work on typestates for objects [5].

In our approach, developers can associate objects with a *hierarchy* of typestates, similar to Statecharts [8]. For example, we will use typestates to indicate whether a Composite node's subtree has an even or odd number of nodes.

Methods correspond to state transitions and are specified with *access permissions* that describe not only the states required and ensured by a method but also how the method will access the references passed into the method. We distinguish exclusive (unique), exclusive modifying (full), read-only (pure), immutable, and shared access (table 1). Furthermore, permissions will specify the *data group* [10] they give access to. Data groups represent orthogonal (logically independent) parts of an object's state. Thus, we can track permissions separately for each data group. We associate a set of mutually exclusive typestates with each data group and therefore will often refer to data groups as *state dimensions*. We use sans-serif all-uppercase words for data groups and all-lowercase words for states. Permissions can optionally include the state the data group is known to be in.

Permissions can only co-exist if they do not violate each other's assumptions. Thus, the following aliasing situations can occur for a given object: a single reference (unique), a distinguished writer reference (full) with many readers (pure), many writers (share) and many readers (pure), and no writers and only readers (immutable and pure).

Permissions are linear in order to preserve this invariant. But unlike linear type systems [11], they allow aliasing. This is because permissions can be *split* when aliases are intro-

Access through other permissions	Current permission has ... Read/write access	Read-only access
None	unique	unique
Read-only	full	immutable
Read/write	share	pure

Table 1: Access permission taxonomy

duced. For example, we can split a unique permission into a full and a pure permission to introduce a read-only alias. Using *fractions* [4] we can also *merge* previously split permissions when aliases disappear (e.g., when a method returns). This allows recovering a more powerful permission.

Fractions are conceptually rational numbers between zero and one. In previous work, fractions below one make objects immutable; in our approach, they can alternatively indicate shared modifying access. Splitting a permission into two means to replace it with two new permissions whose fractions sum up to the fractions in the permission being replaced. Merging two permissions does the opposite. We will sometimes use permissions such as immutable(x, WEIGHT, $1/2$), which represents a permission with exactly a half fraction. Merging two of these permissions yields a full(x, WEIGHT), which gives exclusive modifying access to the WEIGHT data group but still permits pure(x, WEIGHT) permissions to the same object at the same time.

To specify invariants and method pre- and post-conditions we combine permissions (and other atomic predicates such as a variable being **true**) with linear logic operators. We will use multiplicative conjunction ($\otimes$) when two predicates must hold at the same time. Additive conjunction (&) allows internal choice between two predicates, while disjunction ($\oplus$) represents external choice. Linear implication ($\multimap$) will be used when one predicate indicates another.

3. COMPOSITE SPECIFICATION

We now turn to our sample `Composite` class, shown in figure 1, which is a simple implementation of the Composite pattern [7]. Every node in a Composite object tree will be represented by a `Composite` object. The following subsections summarize goals and assumptions before we discuss the class's invariants and method specifications.

3.1 Specification Goals

- Allow clients to add children to any node in a tree.
- Allow nodes to depend on their children in invariants.
- Ensure that adding children to a node does not violate its parents' invariants.

3.2 Assumptions

In order to keep the presentation manageable we use a simplified implementation. We assume that every node in the tree can only have up to two children. We also restrict our discussion to an extremely simple invariant: every node in the tree tracks whether its subtree contains an even or odd number of nodes (including the node itself). This allows our specification to remain in the realm of typestate. Furthermore, notice that `Composite` is the only type of object allowed in a Composite tree. Leafs in the tree are simply `Composite` objects with no children. This lets us ignore problems with inheritance for now. Finally, this specification assumes single-threaded execution.

We discuss extensions to more children and more sophisticated invariants in section 5. Our approach can be extended to include inheritance [3] and multi-threaded programs [2], but these extensions are beyond the scope of this paper.

3.3 Invariants

The focus of our Composite specification is the definition of internal invariants that allow verifying that all nodes in a Composite tree remain consistent when children are added to a node in the tree.

We define 4 data groups [10] and distinguish 2 states in each data group using the **states** keyword.

- The WEIGHT data group defines states that reflect the invariant tracked by our Composite objects: whether the number of nodes in the subtree is even or odd.
- The LEFT (RIGHT) data group each define states that indicate whether the left (right, respectively) subtree contains an even or odd number of nodes (excluding the current node).
- The PARENT data group distinguishes whether the node is an orphan (no parent) or not.

Each of our 4 data groups holds one of the fields defined in the `Composite` class. A permission for the WEIGHT data group, for example, therefore permits access to the *odd* field that it contains, but not the other fields. A full permission gives exclusive write access to the fields in the data group, while an immutable permission gives read-only access with the guarantee that the field will not be (silently) modified.

Unfortunately, however, the fields in the `Composite` class are interdependent in certain ways. In particular, the WEIGHT dimension depends on the objects referenced by the *left* and *right* fields. More precisely, it depends on whether the left and right subtrees contain an odd or even number of nodes. We will model these dependencies as permissions to a data group held by another data group. Figure 2 illustrates the permissions between a node, its parent, left child, and hypothetical client, following the specification in figure 1.

Our intuition for defining invariants is now to use immutable (or full) permissions in an invariant whenever it *depends* on the object or data group referenced with the permission. For example, the WEIGHT data group holds immutable permissions to the LEFT and RIGHT data groups in order to depend on those data groups' states. LEFT and RIGHT, in turn, hold immutable permissions to their children's WEIGHT data groups, which allows LEFT and RIGHT to depend on the children being even or odd.

Based on this structure we can define invariants separately for each data group and their states, which are marked with the keyword **invariant** and the name of the data group or state. Invariants for data groups must always hold, while an invariant for a state defines the condition under which the object is in that state. For readability, we sometimes define multiple **invariant** clauses for the same data group. All invariants defined for a data group must hold at the same time.

Following our intuition, invariants for a data group or state should only mention fields and states that are (transitively) reachable through immutable permissions from the

```
final class Composite {
states PARENT = { orphan, hasParent }
states WEIGHT = { even, odd }
states LEFT = { lefteven, leftodd }
states RIGHT = { righteven, rightodd }

boolean odd; in WEIGHT;
Composite parent; in PARENT;
Composite left; in LEFT;
Composite right; in RIGHT;

invariant PARENT: immutable(this, WEIGHT, 1/2) ⊗ parent ≠ this;
invariant PARENT: (parent = null ⊸ immutable(this, WEIGHT, 1/2)) &
                  (parent ≠ null ⊸ (share(parent, PARENT) ⊗ (immutable(parent, LEFT, 1/2) ⊗ parent.left = this) ⊕
                                                            (immutable(parent, RIGHT, 1/2) ⊗ parent.right = this)));
invariant orphan: parent = null;
invariant hasParent: parent ≠ null;
invariant WEIGHT: immutable(this, LEFT, 1/2) ⊗ immutable(this, RIGHT, 1/2);
invariant WEIGHT: (odd = false ⊸ ((this in leftodd ⊗ this in righteven) ⊕ (this in lefteven ⊗ this in rightodd))) &
                  (odd = true ⊸ ((this in lefteven ⊗ this in righteven) ⊕ (this in leftodd ⊗ this in rightodd)));
invariant even: odd = false;
invariant odd: odd = true;
invariant LEFT: left ≠ null ⊸ immutable(left, WEIGHT, 1/2);
invariant lefteven: left = null ⊕ (left ≠ null ⊗ left in even);
invariant leftodd: left ≠ null ⊗ left in odd;
invariant RIGHT: right ≠ null ⊸ immutable(right, WEIGHT, 1/2);
invariant righteven: right = null ⊕ (right ≠ null ⊗ right in even);
invariant rightodd: right ≠ null ⊗ right in odd;

Composite()
   ensures full(this, PARENT) in orphan ⊗ pure(this, WEIGHT) in odd ⊗
      immutable(this, LEFT, 1/2) ⊗ immutable(this, RIGHT, 1/2);
{ odd = true; parent = null; left = null; right = null; }

void setLeft(Composite c)
   requires share(this, PARENT) ⊗ immutable(this, LEFT, 1/2) ⊗ share(c, PARENT) in orphan ⊗ c ≠ null ⊗ c ≠ this;
   ensures share(c, PARENT) in hasParent ⊗ c.parent = this;
{
   c.parent = this;
   left = c;
   if(c.odd) {
      odd = ! odd;
      Composite p = parent;
      while(p != null) {
         p.odd = ! p.odd;
         p = p.parent;
      }
   }
}

void setRight(Composite c)
   requires share(this, PARENT) ⊗ immutable(this, RIGHT, 1/2) ⊗ share(c, PARENT) in orphan ⊗ c ≠ null ⊗ c ≠ this;
   ensures share(c, PARENT) in hasParent ⊗ c.parent = this;
{ c.parent = this; right = c; if(c.odd) { ... } }

boolean odd()
   requires pure(this, WEIGHT);
   ensures pure(this, WEIGHT) ⊗ ((result = true ⊸ this in odd) & (result = false ⊸ this in even));
{ return odd; }
}
```

Figure 1: Simple `Composite` class with invariants and method specifications

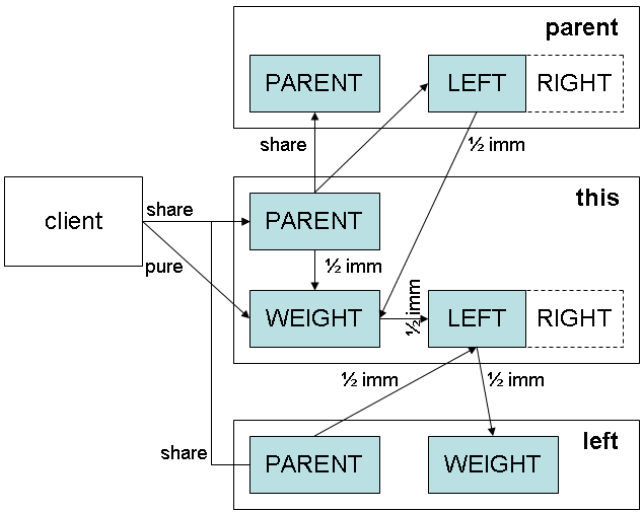


Figure 2: A sample Composite object with parent, left child, and a client that references it. Arrows are labeled with the permissions they represent. Shaded boxes represent data groups inside objects. Only relevant data groups of parent and child are shown.

data group being defined. That ensures that the needed state or field value cannot change without the knowledge of the data group mentioning the state or field in its invariant. For instance, the invariant for the lefteven state includes the *left* field, which is part of lefteven's data group, LEFT, and the state of the WEIGHT data group in the object referenced through the *left* field, for which LEFT holds an immutable permission.

We frequently use internal choice (&) between linear implications (⊸) to encode situations where an *indicator* predicate implies additional facts. For example, the WEIGHT dimension uses the *odd* field as an indicator for states of other dimensions. Internal choice captures the intuition that only one of the indicating predicates can be true at the same time. For example, the *odd* flag cannot be **true** and **false** at the same time.[1] States inside data groups frequently just assert the truth of an indicating predicate.

The PARENT dimension is intended to be used by clients for adding children to nodes. Therefore, we will give out share permissions to this dimension, which allow free modification from multiple places. But in order to add children and modify the *odd* flag, the PARENT dimension holds a permission to the WEIGHT dimension (which in turn references LEFT and RIGHT). It also references the node's parent, if any. The invariants declared for the PARENT dimension are mostly for verification purposes and will be discussed in section 4.

3.4 Method Specifications

The specifications for the `Composite` methods follow from the invariants we discussed above as well as which data groups are accessed in each method. We define the method pre- and post-condition with the **requires** and **ensures** keywords, respectively.

[1]In Boolean logic, internal choice would be expressed as a regular conjunction.

The constructor creates a brand-new `Composite` object without children or parent. The absence of a parent is indicated with the state orphan in the PARENT dimension. The ability to add children comes from the returned immutable permissions for LEFT and RIGHT, which are consumed (i.e., required but not ensured) by the methods for setting the left and right child. Additional immutable permissions for LEFT and RIGHT are kept in the invariant for WEIGHT, as discussed in the previous section. We could also define a method for removing a child, which would return the respective immutable permission to the client. We chose to keep permissions for adding children with the client in order not to have to track them with more `Composite` invariants. Finally, we return a pure permission for the WEIGHT from the constructor, which clients can use to query the *odd* flag with the `odd` method. (Notice that the constructor starts out with full permissions to all data groups, some of which are immediately consumed to satisfy the new object's invariants, resulting in the declared post-condition.)

Setting the left or right child (with `setLeft` and `setRight`, respectively) requires the respective immutable permission for the LEFT or RIGHT dimension in addition to a share permission for the receiver's PARENT dimension. It also requires a share permission for the child's PARENT dimension. The child is required to be an orphan. The invariant for that state linearly implies an immutable permission for the child's WEIGHT dimension, which will be given to the new parent's LEFT (or RIGHT) dimension. In return, the child will capture the given permissions for the receiver in its invariant for hasParent, its ensured state.

Finally, the `odd` method can be used by clients to query whether a node is even or odd. We use a pure permission in the specification of this method. The post-condition uses a linear implication in a way similar to what we discussed for invariants in the previous section: the return value indicates the state of the receiver. The pure permission used in the specification for `odd` implies that the receiver's state can change without the client noticing it. (This is in contrast to immutable permissions, which exclude this possibility.)

One disadvantage of this specification is that once a node has children, the client only has a share permission to the node's PARENT dimension. This is because the initial full that is ensured by the constructor will have to be split into share permissions in order to give some of them to the node's children, as discussed above. Afterwards, clients will loose track of whether a node is an orphan or not. Therefore, our `Composite` probably should have a method `isOrphan` that can be called to test whether a node is in the orphan state. Alternatively, it might be possible to specify the class with an additional dimension that children can use internally to access their parents.

4. IMPLEMENTATION VERIFICATION

This section outlines how the specification presented above can be used to verify the implementation of the `setLeft` method for setting a node's left child. Our verification approach relies on a packing/unpacking methodology which we adapted from existing work [5, 1]. Unpacking a data group releases permissions guaranteed by invariants; packing will consume permissions required by invariants.

Figure 3 shows the `setLeft` method from figure 1 with **pack** and **unpack** commands inserted. Our unpacking *focuses* [6] on the unpacked data group [3] and makes the

```
void setLeft(Composite c)
   requires share(this, PARENT) ⊗ immutable(this, LEFT, 1/2) ⊗ share(c, PARENT) in orphan  ⊗ c ≠ null ⊗ c ≠ this;
   ensures share(c, PARENT) in hasParent ⊗ c.parent = this;
{
   unpack(c, PARENT);
   c.parent = this;
   unpack(this, PARENT);
   if(parent != null)
   { unpack(parent, PARENT); unpack(parent, WEIGHT); unpack(parent, LEFT); unpack(parent, RIGHT); }
   unpack(this, WEIGHT); unpack(this, LEFT);
   left = c;
   pack(this, LEFT); unpack(c, WEIGHT);
   if(c.odd) {
      pack(c, WEIGHT); pack(c, PARENT);
      odd = ! odd;
      pack(this, WEIGHT); if(parent != null) { pack(parent, LEFT); pack(parent, RIGHT); } pack(this, PARENT);
      Composite p = parent;
      while(p != null) {
         if(p.parent != null)
         { unpack(p.parent, PARENT); unpack(p.parent, WEIGHT); unpack(p.parent, LEFT); unpack(p.parent, RIGHT); }
         p.odd = ! p.odd;
         pack(p, WEIGHT); if(p.parent != null) { pack(p.parent, LEFT); pack(p.parent, RIGHT); } pack(p, PARENT);
         p = p.parent;
      }
   } else {
      pack(c, WEIGHT); pack(c, PARENT); pack(this, WEIGHT);
      if(parent != null) { pack(parent, LEFT); pack(parent, RIGHT); pack(parent, WEIGHT); pack(parent, PARENT); }
      pack(this, PARENT);
   }
}
```

Figure 3: Verification of the `setLeft` method from figure 1

invariant of the unpacked data group available as-is even when unpacking a share permission. This means that in order to be sound, we cannot unpack a data group of an object if it is already unpacked. Unpacking the same data group of two references x and y is only allowed when $x \neq y$. This is why we require nodes to be different from their children in the Composite specification (figure 1).[2]

In the `setLeft` method we first unpack the new child, c. Since c is an orphan, we get a full permission to its WEIGHT dimension. Assigning **this** as c's parent will later require the immutable receiver permission from the method pre-condition when packing c. If the receiver has a parent then we need to unpack the permissions we have for the parent[3] in order to gain a full permission for the receiver's WEIGHT dimension (if the receiver is an orphan then that permission is part of its own PARENT invariant). At this point it is crucial that the parent points back to the receiver with its *left* or *right* field: this lets us combine the receiver's own immutable permission to its WEIGHT dimension with the one held by the parent.

Unpacking the full WEIGHT permission for the receiver yields a permission for LEFT, which we also unpack in order to assign c as the receiver's left child. Re-packing LEFT consumes an immutable permission for c's WEIGHT dimension. That leaves another immutable permission for packing c, which we can do right after testing c's *odd* flag. We can update the receiver's *odd* flag—which may have to be changed due to the new child—because we unpacked a full permission for the receiver's WEIGHT dimension, as discussed above.

After updating the receiver's *odd* flag we have to loop through its (transitive) parents to update their flags. Notice that only two objects are unpacked at a time: the object pointed to by p and its immediate parent. We can unpack both of them because p's invariants guarantee that it is distinct from its parent. We do *not* prevent cycles in the Composite tree with our specification, but since we only unpack two objects at a time we can still verify partial correctness.

Updating the parents proceeds similarly to updating the receiver (without assigning *left* or *right*). One interesting issue is that when we unpack $p.parent$, we only have an immutable($p.parent$, WEIGHT, 1/2) permission available. Later on, in the next loop iteration, *its* parent is unpacked (or we discover that is has no parent), which yields a second immutable permission, giving us full(p, WEIGHT). That allows us to to update the *odd* flag and re-pack.

[2]We previously only allowed one data group in one object to be unpacked at a time [3], but we believe that the more permissive rule described here preserves soundness.

[3]We unpack both the parent's LEFT and RIGHT dimension because a child does not know if it is the left or right child.

5. FUTURE WORK

This section discusses limitations of the Composite specification presented in section 3 and how they can be overcome.

Non-typestate invariants. Because our approach focuses on typestates we have chosen an invariant, even vs. odd, that can be expressed with typestates. But we believe that the presented specification can be adapted for other invariants specified in the WEIGHT dimension. For example, if nodes wanted to track the number of nodes in their subtree, they could declare a field *weight* and define the invariant $weight = lw() + rw() + 1$, where *lw* and *rw* are functions that return the number of nodes in the left and right subtree, respectively. Similar to subtrees remaining even or odd, these numbers are guaranteed to remain valid because WEIGHT is relying on them using immutable permissions. Notice that an invariant based on the number of nodes would shorten our specification substantially because we would not need to define the meaning of "even" and "odd" explicitly. Verifying such properties may require a theorem prover to reason about integers, which we believe could be added to our approach.

Many children. An arbitrary number of children can for example be achieved with a list or array. In order to specify invariants over this list, we will need to describe the invariant to the child held in each list element or array cell. Moreover, we will need an invariant for the children that guarantees the parent to point back to them. It appears that one could put each list element or array cell into a separate data group (with separate permissions), similar to the LEFT and RIGHT groups, and we are working on a way of supporting this in specifications.

Method calls while unpacked. We put the code for adding a child into a single method that contains a loop to iterate through the receiver's parents (see figure 3). It would be nicer to, for example, call a method on the new child to set its parent, and to call a method on the parent to update its invariant. The presented implementation was chosen because objects involved in these calls would be unpacked at the call sites. Moreover, it is harder to guarantee that no data group of any object is unpacked more than once when multiple methods may unpack objects at once. We believe that a specification language similar to Spec# [1], which can specify objects to be unpacked at method boundaries, could remove this restriction.

Precise effects. As discussed, the pure permission used for specifying the odd method (figure 1) reflects that state changes in the WEIGHT dimension can happen without the client noticing. But while our approach will "forget" whether a node was even or odd upon any effectful operation, more precise tracking of effects may enable forgetting this information only if nodes are added to the node's subtree (which is when the node's state actually changes).

Overhead reduction. The specification overhead in figure 1 is arguably high. About half of the invariants relate to the particular property we are tracking, even vs. odd. The rest represents a pattern for encoding circular dependencies between objects with immutable permissions. We believe that a developer could reuse this pattern to track the property of interest on top of it. This suggests introducing a specification construct for defining circular dependencies, which would internally be translated into the invariants shown, to reduce specification size.

6. CONCLUSIONS

This paper presents a specification of the Composite design pattern with permissions that allows nodes to depend on their children in invariants and allows clients to add children to any node in a Composite tree at any time. Permissions can express the circular dependencies between nodes that are needed to guarantee that adding a child to a node correctly updates the parents' invariants. We discuss how shortcomings of the presented specification can be overcome with a richer specification language than the one used in this paper. In particular, we believe that our permission based approach can be extended from typestate-based to more interesting invariants and to arrays or lists of objects.

Acknowledgments. We thank Nels Beckman and the anonymous reviewers for their helpful feedback on earlier versions of this paper. This work was supported in part by the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", DARPA contract HR00110710019, and NSF grant CCF-0811592.

7. REFERENCES

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[2] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of Atomic blocks and typestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2008. To appear.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 301–320, Oct. 2007.

[4] J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.

[5] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

[6] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[9] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, submitted for publication.

[10] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998.

[11] P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

Model Programs for Preserving Composite Invariants

Steve M. Shaner
Iowa State University
smshaner@cs.iastate.edu

Hridesh Rajan
Iowa State University
hridesh@cs.iastate.edu

Gary T. Leavens
University of Central Florida
leavens@eecs.ucf.edu

ABSTRACT

We describe a solution for the SAVCBS challenge problem: a technique for specifying and verifying invariants for objects designed using the Composite design pattern. The solution presents a greybox specification technique using JML's model program feature. We show that model program specifications function as exemplars for capturing helper method calls in a way that preserves modularity and encapsulation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement — Documentation; D.2.11 [**Software Engineering**]: Software Architectures — Information hiding, Languages, Patterns; D.3.3 [**Programming Languages**]: Language Constructs and Features — classes and objects, inheritance; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques.

General Terms

Verification

Keywords

Greybox specification, verification, model program, composite design pattern, JML language.

1. INTRODUCTION

Invariants in formal specification languages, such as the Java Modeling Language (JML) [5, 7, 8] describe relationships that hold in each visible object state. They both document intended relationships and impose proof obligations on code for each object. Our solution to the 2008 SAVCBS challenge problem describes a methodology for verifying invariants of objects designed using the Composite pattern [6, pp. 163]. We demonstrate how model program specifications can be used to enforce a simple invariant for Composite objects [9, Section 4.1].

A model program [16] is JML's realization of the greybox specification technique [4]. A model program is thus a hybrid between program code and specification, and can be thought of as an abstract algorithm [1, 2, 3, 11, 12]. The algorithm is abstract in the sense that it may suppress many implementation details, only specifying their effects using specification statements. This allows the specifier to hide some details, while showing others to the reader. In the refinement calculus, code satisfies an abstract algorithm specification if the code refines the specification [1, 12, 11]. However, JML currently limits refinement of model program by requiring all exposed code to match the implementation exactly [16].

A major benefit of JML's model programs is that they can be used to specify functional dependencies among objects in a straightforward way—by exposing code that maintains the dependencies. In this paper we show how this technique works within the Composite class of the Composite design pattern. For complex object structures like those seen in the Composite design pattern, a set of helper methods can be defined that exploit the object structure to establish the invariant. Writing model programs that show how all methods that may affect the invariant invoke these helper methods and then showing that these model programs individually preserve the invariant is our recommended methodology. We demonstrate this methodology in our solution to the challenge problem.

2. JML MODEL PROGRAMS

Model program specifications in JML, like their counterpart in Büchi and Weck's greybox specifications [4], are algorithmic abstractions of concrete functionality. They selectively expose only the desired parts of a method's concrete behavior. In particular they can specify calls of certain methods in specified states; we call such specified method calls "mandatory" calls [16].

Model program specifications have two major benefits:

1. A suitable model program specification allows more expressive invariants on the concrete behavior compared to a behavioral specification, and
2. Such invariants do not depend upon hidden implementation details, thus they improve information-hiding modularity [15] compared to exposing all of the implementation for the purpose of writing invariants.

Figure 1 shows an example JML specification for the class `ElementCollection`. In the special JML comments, the private field `inner` is declared to be public for specification purposes using **`spec_public`**. The model program specification for

the addAll method for ElementCollection class is shown in Figure 1 on lines 4–13. This specification has two parts: a behavioral specification statement on lines 5–10 and a white-box specification on lines 11 and 12. Behavioral specification statements all begin with the **normal_behavior** keyword and this one contains a precondition (the **requires** clause), a frame axiom (**assignable** clause), and two postconditions (**ensures** clauses). In postconditions the operator \old is used to refer to the previous state. The **normal_behavior** keyword that starts specification statements selects a total correctness specification that allows no exceptions to be thrown; i.e., if execution starts in a state that satisfies the precondition, then it must terminate normally in a state that satisfies both the frame axiom and the postconditions.

```
1 class ElementCollection extends Collection {
2   private /*@ spec_public @*/
3     Collection inner;
4   /*@ public model_program {
5     @   normal_behavior
6     @     requires inner != null;
7     @     assignable this.inner;
8     @     ensures c.size() == \old(c.size());
9     @     ensures this.inner.size() ==
10    @       \old(this.inner.size());
11    @   for (Element e : c)
12    @     this.add(e);
13    @ } @*/
14  public void addAll(ElementCollection c) {
15    /*@ refining normal_behavior
16      @   requires inner != null;
17      @   assignable this.inner;
18      @   ensures c.size() == \old(c.size());
19      @   ensures this.inner.size() ==
20      @     \old(this.inner.size()); @*/
21    { /* resize array if necessary */ }
22    for (Element e : c)
23     this.add(e);
24  } }
```

Figure 1: An example JML model program specification.

The behavioral specification describes invariants maintained by the parts of the concrete implementation that are not visible in the specification. This hides changeable implementation details from the client code. The white-box specification exposes part of the concrete implementation to allow clients to write more expressive invariants. A simple example of such increased expressiveness would be the guarantee that all invariants maintained by the method add for class ElementCollection (not shown) will also be preserved by the method addAll. Thus, for example if the method add maintains a count of all elements in the collection, such count would be accurate event if elements are added in bulk using the method addAll.

Such model program specifications are only valid for use in reasoning if the concrete implementations refines them [1, 11, 12]. For JML model program specifications instead of adopting a general notion of refinement, a more pragmatic approach based on structural refinement is adopted. A concrete implementation refines a model program specification if it is structurally the same as the specification, up to the code implementing the black-box specification behaviors. In Figure 1 the concrete implementation declares that the elided code on line 21 refines the black-box specification on lines 15–20 using the **refining** keyword. Each **refining** statement must have the same specification part and a body (between the braces and outside the special JML comments) that satisfies that specification. This makes verifying refinements easier. The rest of the method implementation (lines 22 and 23) is identical to its counterpart in the specification (lines 11 and 12). The details of the refinement technique are described in detail by Shaner, Leavens and Naumann [16].

In addition to the semantics described above [16], in this paper we also require that a correct implementation of a specification statement must not call any methods that are explicitly called in the whitebox (executable) code portion of the model program. We will see why this is needed below.

Since each **refining** statement contains a specification, writing model program specifications separate from the concrete implementation is often verbose and redundant. To reduce annotation burden and the cost of keeping model programs consistent with respect to the concrete implementation, JML also provides an additional syntactic sugar **extract** for extracting such specifications. An example of this feature is shown in Figure 2.

```
1 class ElementCollection extends Collection {
2   private /*@ spec_public @*/
3     Collection inner;
4   public /*@ extract @*/
5    void addAll(ElementCollection c) {
6     /*@ refining normal_behavior
7       @   requires inner != null;
8       @   assignable this.inner;
9       @   ensures c.size() == \old(c.size());
10      @   ensures this.inner.size() ==
11      @     \old(this.inner.size()); @*/
12    { /* resize array if necessary */ }
13    for (Element e : c)
14     this.add(e);
15  } }
```

Figure 2: Extracting Model Program Specifications.

In this version of the method addAll the **extract** keyword is used on line 4. This results in an automatic generation of the model program specification during verification. The automatic extraction of model program specification proceeds by suppressing the bodies of **refining** statements, replacing them with the specifications they contain as specification statements. The extracted model program specification for our example is the **normal_behavior** statement found on lines 6-11 in Figure 1, followed by the **for**-loop exactly as it appears in the code.

3. COMPOSITE'S SPECIFICATION

Our solution, given in Figure 3 and Figure 4, contains a combination of model programs, helper methods and pure methods.

The class Component is specified in Figure 3. As in previous example, the two protected fields parent and total are declared to be public for specification purposes using **spec_public**. The invariant in this figure is not the one we are mainly concerned with.

The subclass Composite is specified in Figure 4. It has 2 fields: an array components and an integer count. Lines 8–11 give the invariant we are mainly concerned with for the challenge problem; it states that total is one more than the sum of the values of the total fields of each object in the components array.

The two methods in Figure 4 have model program specifica-

```
1 class Component {
2   protected /*@ spec_public nullable @*/
3     Composite parent;
4   protected /*@ spec_public @*/ int total = 1;
5   //@ protected invariant 1 <= total;
6 }
```

Figure 3: Specification of Component.

tions: `addComponent` and `addToTotal`. The model program for method `addComponent` is given explicitly on lines 13–24 (preceeding that method's header), while the model program for `addToTotal` is implicit in the body of the method `addToTotal`, as indicated by the keyword **`extract`** [16]. The automatically extracted model program specification, for the method `addToTotal` is shown in Figure 5.

3.1 Specification

We now describe how the model programs of Figure 4 specify that `Composite` instances preserve the invariant on lines 8–11.

Consider method `addComponent`. The primary responsibility of this method is to modify the representation array `components` and appropriately update the `total` field. The invariant adds a subtle complexity to this update by requiring that the value of each subcomponents' `total` field is included in the value of its parent's `total` field. Thus a correct implementation of `addComponent` must capture the structural relationship between the composite and its subcomponents and use this information when updating the `total` fields.

In our example, this structural relationship is captured by the definition of method `addToTotal`. It both modifies this instance's `total` field and asks that the parent (if one exists) be modified as well. This has the useful effect of re-establishing the invariant for all instances for which the invariant might have been violated, provided `addToTotal` is called only once, and with the appropriate argument.

For this problem, we have written a model program for `addComponent` that exposes its call to `addToTotal`. Recall that, due to the restricted notion of refinement in our technique, correct implementations of `addComponent` must call `addToTotal` after changing both parent and child, as described by the model program. It is this notion of "structural similarity" that makes the call to `addToTotal` "mandatory" [16]. In proving that a model program for `addComponent` is refined by its implementation, we show structural similarity between the model program and the implementations of `addComponent` in all subclasses of `Composite`. Thus, if the model program preserves the invariant for all `Composite` objects, then the invariant will be preserved by all subclasses, since they must also refine the model program in the same sense.

As noted above, we require that specifcation statements must not call any methods that are explicitly called in model program. For the specifications in Figure 4, this means that the bodies of the **`refining`** statements inside the implementation of `addComponent` are prohibited from calling `addToTotal`.

```
1 class Composite extends Component {
2   private /*@ spec_public @*/
3     Component[] components = new Component[5];
4       //@ in objectState;
5       //@ maps components[*] \into objectState;
6   private /*@ spec_public @*/ int count = 0;
7       //@ in objectState;
8   /*@ protected invariant total
9     @       == 1 + (\sum int i;
10    @               0 <= i && i < count;
11    @               components[i].total); @*/

13  /*@ public model_program {
14    @  normal_behavior
15    @     requires c.parent == null;
16    @     assignable this.components;
17    @     ensures this.components.length
18    @             > this.count;
19    @  normal_behavior
20    @     assignable c.parent, this.objectState;
21    @     ensures c.parent == this;
22    @     ensures this.hasComponent(c);
23    @  addToTotal(c.total);
24    @ } @*/
25  public void addComponent(Component c) {
26    /*@ refining normal_behavior
27      @   requires c.parent == null;
28      @   assignable this.components;
29      @   ensures this.components.length
30      @         > this.count; @*/
31    { /* resize components, if necessary */ }
32    /*@ refining normal_behavior
33      @   assignable c.parent, this.objectState;
34      @   ensures c.parent == this;
35      @   ensures this.hasComponent(c); @*/
36    {
37      components[count] = c;
38      count++;
39      c.parent = this;
40    }
41    addToTotal(c.total);
42  }
43  private /*@ helper extract @*/
44    void addToTotal(int p) {
45    /*@ refining normal_behavior
46      @   requires 0 <= p;
47      @   assignable this.total;
48      @   ensures this.total
49      @        == \old(this.total) + p; */
50    { total += p; }
51    Component aParent = this.parent;
52    while (aParent != null) {
53      /*@ refining normal_behavior
54        @   assignable aParent.total, aParent;
55        @   ensures aParent.total
56        @        == \old(aParent.total) + p;
57        @   ensures aParent
58        @        == \old(aParent.parent); @*/
59      {
60        aParent.total += p;
61        aParent = aParent.parent;
62  } } }
63  /*@ pure @*/ boolean hasComponent(Component c) {
64      // ...
65 } }
```

Figure 4: JML model program specification for Composite, based on Leavens, Leino, and Müeller's specification [9, Figure 10].

```
1  /*@ public model_program {
2    @   normal_behavior
3    @     requires 0 <= p;
4    @     assignable this.total;
5    @     ensures this.total
6    @          == \old(this.total) + p;
7    @   Component aParent = this.parent;
8    @   while (aParent != null) {
9    @     normal_behavior
10   @       assignable aParent.total, aParent;
11   @       ensures aParent.total
12   @            == \old(aParent.total) + p;
13   @       ensures aParent = aParent.parent;
14   @   }
15   @ } @*/
16 void addToTotal(int p);
```

Figure 5: Extracted specification for `addToTotal`.

4. PROBLEMS & SOLUTIONS

The model programs of Figure 4 assist reasoning with invariants in two scenarios of interest: handling argument exposure for Composite clients, and when defining Composite subclass methods. We break the latter problem down into two mutually exclusive subproblems, the overriding of existing methods and the introduction of new ones.

4.1 Argument Exposure in Client Reasoning

Argument exposure occurs when an invariant, such as the one in Figure 4, depends on objects that are not under control of the object's methods [14]. In that figure, the invariant of `Composite` depends on the components in the `components` array. The challenge is how to maintain such an invariant when clients may change objects on which the invariant depends without calling methods directly on the object.

Let us consider how our specification in Figure 4 and the greybox approach (JML's model program technique) deal with this problem. In essence our solution is a special case of the visibility technique for maintaining invariants [9, 13]. To see this, note that the fields `total`, `components`, and `count` cannot be written by classes that do not see the invariant in Figure 4, because these fields are protected and private and the invariant is protected. Hence the invariant can be maintained in each subclass of `Composite`, by requiring all these subclasses to maintain it each time they change one of these three fields.

The key point of our specification is that the model program and the code it requires follow the chain of `parent` links upwards, and adjusting each total of each parent object. Since the precondition of `addComponent` requires that `c.parent` be null, no cyclic or aliased structure can be created using `addComponent`, thus there is always at most one parent for each Component `c`.

To see how this is done, consider the client code in Figure 6. This sets up the problematic case of a Composite object, `root`, that contains another Composite object, `child`, which itself contains the component `comp`. If `addComponent` maintains the invariant, then the assertion at the end of the figure should hold, even though line 12 modifies `child` without calling a method on its parent `root`. The invariant should apply when reasoning about the resulting heap structure, regardless of the order in which the components get added to each other.

```
1       Composite root = new Composite();
2       Composite child = new Composite();
3       Component comp = new Component();

5       //@ assume root.total == 1;
6       //@ assume child.total == 1;
7       //@ assume comp.total == 1;
8       //@ assume root.parent == null;
9       //@ assume child.parent == null;
10      //@ assume comp.parent == null;
11      root.addComponent(child);
12      child.addComponent(comp);
13      //@ assert root.total == 3;
```

Figure 6: Clients reason by instantiating invariants for concrete contexts like this one, in which a tree of three components is built.

The model programs described in Figures 4 and 5 are used in verification by substituting the model program's body for the call site of the method it specifies (with actuals replacing formals and care taken to avoid capture). In Figure 6, this means substituting in lines 14–23 of Figure 4 for each call to `addComponent`, renaming occurrences of the formals `c` and **`this`** to the appropriate instances. Furthermore, each of these substitutions exposes a call to `addToTotal`, so its model program body can be substituted similarly.

The resulting code resembles Figure 7. In this figure, lines 11–34 are the model program for `addComponent` substituted for the call on line 11 of Figure 6. Similarly, lines 35–58 are for the call on line 12 in the original. From this text and a Hoare-style proof system, we can verify that the closing assertion holds. This proof is straightforward after assuming the proof rules given in previous work [16] with a standard extension to handle while loops.

4.2 Overriding Composite's Methods

In subclasses of `Composite`, a developer might incorrectly try to override its methods `addComponent` or `addToTotal` in a way that violates the invariant or a model program specification. However, such an override would be incorrect in our technique, because not only are invariants inherited by subtypes in JML [7], but subtypes also inherit model program specifications. Thus methods inheriting a model program are subject to the same structural constraints as the overridden method. Though subclass implementors are free to refine the bodies of **`refining`** statements as long as they satisfy the contract behavior, all other exposed code must appear as it does in the model program. In this fashion, as long as the original model program preserves the invariant, subclass overrides of those methods cannot violate the invariant.

4.3 Extending Composite with New Methods

Subtypes also pose problems when they introduce new methods that do not override methods in their supertype(s). Such methods must preserve the inherited invariants, as would be the case in our example, but our technique does not yet provide direct support for this situation.

In our example, the way in which `Composite`'s invariant is maintained depends heavily on two assumptions: (a) `addComponent` is the only method that adds components to a composite, and (b) `addComponent` has a precondition that requires the parent of the added component to be null. An added method could violate these

```
1 Composite root = new Composite();
2 Composite child = new Composite();
3 Component comp = new Component();

5 //@ assume root.total == 1;
6 //@ assume child.total == 1;
7 //@ assume comp.total == 1;
8 //@ assume root.parent == null;
9 //@ assume child.parent == null;
10 //@ assume comp.parent == null;
11 normal_behavior
12   requires child.parent == null;
13   assignable root.components;
14   ensures root.components.length
15         > root.count;
16 normal_behavior
17   assignable child.parent, root.objectState;
18   ensures child.parent == root;
19   ensures root.hasComponent(child);
20 {
21   normal_behavior
22     requires 0 <= child.total;
23     assignable root.total;
24     ensures root.total
25          == \old(root.total) + child.total;
26   Component aParent = root.parent;
27   while (aParent != null) {
28     normal_behavior
29       assignable aParent.total, aParent;
30       ensures aParent.total
31            == \old(aParent.total) + c.total;
32       ensures aParent = aParent.parent;
33   }
34 }
35 normal_behavior
36   requires comp.parent == null;
37   assignable components;
38   ensures child.components.length
39         > child.count;
40 normal_behavior
41   assignable comp.parent, child.objectState;
42   ensures comp.parent == child;
43   ensures child.hasComponent(comp);
44 {
45   normal_behavior
46     requires 0 <= comp.total;
47     assignable child.total;
48     ensures child.total
49          == \old(child.total) + comp.total;
50   Component aParent = child.parent;
51   while (aParent != null) {
52     normal_behavior
53       assignable aParent.total, aParent;
54       ensures aParent.total
55            == \old(aParent.total) + comp.total;
56       ensures aParent = aParent.parent;
57   }
58 }
59 //@ assert root.total == 3;
```

Figure 7: The client code of Figure 6 after substituting the bodies of the model programs for `addComponent` and `addToTotal` and renaming field references to the appropriate instances.

assumptions, allowing aliased Composite structures to be created that our proof does not handle. Since the invariant about lack of aliasing is not stated explicitly in our specification, it is not clear how this part of our argument would apply to subtypes. To avoid this problem, one would have to write a static (i.e., global) invariant that described the required lack of aliasing. But this fix seems specific to the Composite design pattern, and it is not clear how our technique could be generalized to handle it.

5. DISCUSSION

At the class level, our example model programs describe the set of methods that are responsible for maintaining the representation invariant. They provide an abstract overview of where and how the invariant is maintained. The only way subcomponent membership can change is by calling `addComponent` and the only way `total` is updated is by a call to `addToTotal`. No calls to `addComponent` occur within this class, but if they did, a model program exposing that call could be written.

Model programs enable modular descriptions of the internal structure of code in ways that are useful for client reasoning. By choosing model programs to control the static structure of subclass implementations, our solution relies on the mechanical textual matching described in our previous work [16]. JML's **`refining`** statements clearly identify which specification statements are refined where inside of an implementation, while the **`normal_behavior`** specification statements use pure methods to hide specific representation details.

This is not to say that the working definitions used by our solution are perfect. Adding nondeterministic loops, conditionals and other constructs to the model program syntax would increase flexibility when matching implementations against a model program specification. Also, work on this paper highlighted a number of visibility issues for model programs that have not previously been investigated [10]. A basic issue is defining rules that respect visibility for model program specifications. There is also a complication posed by **`extract`**, which may pull out specifications and code that are legal within a method, but which may refer to private data. If the extracted model program is to be public, then such private data is not understandable by all clients, and so should be disallowed. We finessed this problem in our example by declaring all fields as **`spec_public`**, but this is certainly an area where more work is needed.

6. CONCLUSION

We have described how model program specifications can be used to specify and verify invariants in complex heap data structures created using the Composite design pattern. Our solution is a fruitful combination of the visibility technique for invariants with the greybox specification technique. The combination is fruitful because the greybox technique allows specifiers to describe exactly how a method must update all invariants that might be violated. In our example, `addComponent` is specified to update all of the `total` fields of all parents. This detail is crucial in maintaining the invariant for all `Composite` objects.

Acknowledgments

The authors were supported in part by the NSF grant CNS-06-27354 and CNS 08-08913.

7. REFERENCES

[1] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[2] R. J. R. Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6):593–624, 1988.

[3] R. J. R. Back and J. von Wright. Refinement calculus, part i: sequential nondeterministic programs. In *REX workshop: Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 42–66, New York, NY, 1990. Springer-Verlag.

[4] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999.

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

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Z. Liu and H. Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, Nov. 2006. Springer-Verlag.

[8] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[9] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007.

[10] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, May 2007.

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

[12] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[13] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, Oct. 2006.

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

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–8, Dec. 1972.

[16] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, pages 351–367. ACM, Oct. 2007.