Integrating Static Checking and Interactive Verification: Supporting Multiple Theories and Provers in Verification

Joseph R. Kiniry
Systems Research Group
School of Computer Science and Informatics
UCD Dublin
Belfield, Dublin 4, Ireland

Patrice Chalin
Dependable Software Research Group
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, H3G 1M8, Canada

Clément Hurlin
Université Henri Poincaré, Nancy 1
BP 60120, Nancy Cedex, France

with contributions from
Cees-Bart Breunesse, David Cok, Bart Jacobs,
Erik Poll, Silvio Ranise, Aleksy Schubert, and Cesare Tinelli

Abstract

Automatic verification by means of extended static checking (ESC) has seen some success in industry and academia due to its lightweight and easy-to-use nature. Unfortunately, ESC comes at a cost: a host of logical and practical completeness and soundness issues. Interactive verification technology, on the other hand, is usually complete and sound, but requires a large amount of mathematical and practical expertise. Most programmers can be expected to use automatic, but not interactive, verification. The focus of this proposal is to integrate these two approaches into a single theoretical and practical framework, leveraging the benefits of each approach.

1. Introduction

Endemic in society today are problems related to the lack of software quality which, as a result, is costing governments, businesses, and nations billions of dollars annually [15]. Correctness and security issues are also directly related to some of the most important concerns of the day such as those of national security and technology-based voting.

Additionally, driven by governmental regulations and market demands, businesses are now slowly beginning to assume liability for the faults exhibited by the software systems they offer to their customers. This is particularly true in safety and security critical domains.

While a variety of software engineering practices have been developed to help increase software quality (e.g., testing practices, system design, modern processes, robust operating systems and programming languages), it is widely acknowledged that a promising way to achieve highly reliable software in critical domains is to couple these practices with applied formal techniques supported by powerful modern tools and technologies like those discussed in this paper.

1.1. Program Verification

Applied formal methods has turned a corner over the past few years. Various groups in the semantics, specification, and verification communities now have sufficiently developed mathematical and tool infrastructures that automatic

and interactive verification of software components that are written in modern programming languages like Java has become a reality.

Automatic verification by means of Extended Static Checking (ESC) has seen some success in industry and academia due to its lightweight and easy-to-use nature. Unfortunately, ESC comes at a cost: a host of logical and practical completeness and soundness issues. Interactive verification technology, on the other hand, is usually complete and sound, but requires a large amount of mathematical and practical expertise. Typical programmers can be expected to use automatic, but not interactive, verification.

In this paper we discuss work which has been undertaken to:

- integrate the ESC and interactive verification approaches into a single *theoretical framework*, and
- directly realize this theoretical framework in a modern software development environment (IDE) as an Open Source initiative.

Specifically, our current work is focused on the integration of the verification technologies behind two successful tools, namely ESC/Java2 and the Loop program verifier (both will be described shortly). The key idea is to build a single environment whereby as much verification as possible happens automatically, and thus use of interactive verification only happens when necessary. (In those situations where developers wish to delay completion of the interactive proofs, it will be possible to insert run-time assertion checking code to perform compensatory verification during code execution.)

2. Two Key Java Verification Tools

Next, we discuss two complementary verification tools for Java upon which we base this work.

2.1. Extended Static Checking: ESC/Java2

A successful automatic verification tool for Java is ESC/Java, an extended static checker originally developed at DEC SRC [7]. The next-generation release, called "ESC/Java2", is now available as an now available as an Open Source project that is supported by academic and industrial researchers [5]. David Cok and the first author are the ESC/Java2 project administrators and main contributors. ESC/Java2 is currently used as a research foundation by over a half dozen research groups and as an instructional tool in nearly two dozen software-centric courses around the world.

ESC/Java2 processes Java programs that are annotated with the Java Modeling Language (JML) [2, 12]. ESC/Java2 automatically converts JML-annotated Java code into verification conditions that are automatically discharged by an embedded theorem prover—currently, Simplify [6]. Problems in the specifications, programs, or the checking itself are indicated to the user by means of error messages. As ESC/Java2's performance and mode of interaction are comparable to an ordinary compiler, it is quite usable by industry developers as well as computer science and software engineering students.

2.2. Interactive Verification: The Loop Tool

The Loop tool, developed by the SoS Group at Radboud University Nijmegen under the supervision of Prof. Bart Jacobs, is an interactive verification tool for JavaCard [3]. The Loop tool is one of the most complete verifiers with respect to the subset of Java that it covers. Loop compiles JML-annotated Java programs into proof obligations expressed as theories for the PVS theorem prover. By making use of PVS to interactively discharge the proof obligations, he or she is able to prove a program correct with respect to its JML specification.

The base Java/JML semantics of the Loop tool essentially consists of a parameterized theory. The theory parameters are for the (sub-)theory to be used to reason about integral types. Early in the LOOP Project, Java's integral types were modeled by the mathematical integers. Later, support was added for bounded integers (with the familiar modulo arithmetic) and a bitvector representation (which facilitates reasoning about bit-wise operations—something that is common in JavaCard applications). When reasoning about Java programs, one has a choice of program logics including Hoare logics and two weakest precondition calculi. Recently, Breunesse has merged these into a single, unified theory in which different representations can be used simultaneously [1].

As these two tools represent some of the best-of-breed of applied formal methods in the Java domain, integrating their foundations and approaches has merit. To accomplish this goal, there are several theoretical and practical challenges to be faced.

3. Integration: Observations and Challenges

There is no single canonical semantics of Java. The canonical informal semantics for Java is embodied in the Java Language Specification [9]. Various groups have formalized portions of this text and built complementary tools, e.g., the

- Everest Group at INRIA Sophia-Antipolis,
- SoS Group at the Radboud University Nijmegen,

- Logical Group at INRIA Futurs/Université Paris-Sud,
- SAnToS Laboratory at Kansas State University,
- KeY group, composed of researchers from the Chalmers University of Technology, the University of Koblenz, and the University of Karlsruhe,
- Software Component Technology Group at ETH Zürich, and
- now disbanded Extended Static Checking Group at HP/Compaq/Digital Systems Research Center.

In all of these cases the formalizations are incomplete, either in scope or in accuracy.

There is no single, core, canonical semantics of JML. While there are several partial informal and formal semantics for JML, there is no single, core semantics. Furthermore, the informal semantics of JML is much more transient and imprecise than that of Java, so the problems mentioned above for Java are compounded for JML. This state of affairs leads to subtle inconsistencies between the interpretation of specifications by the tools that support JML. Because of this inconsistency, relating the semantics to each other is extremely difficult. Additionally, explaining, extending, and reasoning about these artifacts (e.g., the calculi of ESC/Java2) is very difficult.

Little work has been done on meta-logical reasoning about object logics. By meta-logical reasoning we mean reasoning *about*, rather than *within*, the semantics of program and specification languages. Formal meta-mathematical proofs are rare. It is not known, for example, if ESC/Java2's object logic is sound. This is a critical issue.

4. An Integrated Verification Environment

In collaboration with others, our research groups have begun work on an integrated verification environment (IVE) and its necessary theoretical foundations. In doing so we have started to address the problems identified in the previous section.

We are (concurrently) working on the achievement of the following initial milestones:

- elaboration of a semantics for a "core" JML,
- extracting, analysing, and extending ESC/Java2's logic and calculi, and
- redesigning ESC/Java2's proof infrastructure as well as backend interfaces and adaptors with the main objective of allowing it to support new provers.

4.1. Semantics for "Core" JML

Semantics have been developed for JML within different logics, nearly all of which have been embedded in the various tools developed by the groups enumerated in Section 3. A few of these tools are publicly available, but most were never used outside the group that originally developed them.

To resolve ambiguities, disagreements, and lack of detailed formal documentation within the JML community, a *single, open* semantics of a "core" of JML needs to be written. Chalin and Kiniry are currently outlining a proposed core and have begun formalizing its definition. The outcome of this effort is also a major goal of the MOBIUS project [13].

This semantics will be written in a well-understood formalism, e.g., within a modern extension to Hoare logic, a denotational semantics, and/or in a concurrent operational semantics. In our initial work we have decided to express our base, canonical semantics in PVS and Isabelle. Realizing the object logic within higher-order provers will help us characterize and compare semantics.

It is expected that multiple formalizations of the object logic will be created due to practical and theoretical reasons. E.g., most research groups have developed expertise in only one prover, and furthermore, the community can benefit with experimentation with the varying capabilities of each of these provers.

4.2. Evolving ESC/Java2's Logic & Calculi

As inherited from its predecessor SRC's ESC/Java, ESC/Java2 makes use of an unsorted object logic and two calculi (a weakest precondition calculus and a strongest postcondition calculus). The unsorted object logic consists of approximately 80 axioms written in the language understood (only) by the Simplify prover. These axioms are highly tuned to the quirks and capabilities of Simplify. Recent developments in ESC/Java2 saw the logic extended by another (approx.) 20 axioms.

A transcription of this Simplify-based unsorted object logic has been written in PVS. We refer to this formalization of the logic as EJ_0. Two other logics, EJ_1 and EJ_2, have also been written; EJ_1 is merely a sorted version of EJ_0 whereas EJ_2, also a sorted logic, was written from scratch with the purpose of better representing the abstractions needed by ESC/Java2 to reason about JML annotated Java programs. Soundness proofs as well as results on the (semi-)equivalence of the EJ_i logics are underway.

We will also be "extracting" the weakest precondition and a strongest postcondition calculi of ESC/Java2, as well as at least one of the weakest precondition calculi used with the Loop verification system. This will most likely

be done in a higher-order logic or a term rewriting framework. The rewriting speed of special purpose environments like Maude [4] may be of benefit as the tool and verification efforts scale to larger problems.

4.3. Supporting Multiple Provers

As we progress in our work on the definition and proofs of soundness and completeness of the EJ_i logics, we are also progressing in our work on extending and adapting ESC/Java2 to support multiple provers. By developing a generic prover interface along with suitable adaptors, we plan on experimenting with a few next-generation first-order provers.

We anticipate the possibility of supporting the use of multiple provers, simultaneously or independently. Which prover to use might be determined automatically by ESC/Java2 based on the context of the verification and the capabilities of the provers. For example, while Simplify is a very fast predicate solver, it does not support a complete or sound (fragment of) arithmetic, thus in verification contexts where arithmetic is used, the tool should automatically avoid using Simplify.

We have chosen the Sammy and haRVey provers as the initial provers for experimentation [8, 14]. This choice was made due to our research relationship with the authors of these two tools as well as the authors' high-profile position within the SMT-LIB community [16].

As a necessary precursor to being able to support multiple provers, we are required to translate our object logic, whose current canonical representation is is PVS, into an appropriate formalism understood by each of the provers. Encoding of the ESC/Java2 object logic for these provers is being accomplished primarily by their respective research teams, not by us.

We will also be experimenting with the use of higher-order provers as backends for ESC/Java2. Our initially targeted provers are PVS, Isabelle, and Coq. Aside from the authors, Aleksy Schubert is working on a Coq realization of the object logic whereas Erik Poll is contributing to the PVS realization.

5. Conclusion

One of the advantages of our project is that we have a working toolset today that supports Java and JML. These tools are actively being used by researchers and a few industry practitioners. Our goal is to help evolve these tools into their next-generation counterparts and, all the while, make sure that we take our own medicine.

Thus, for example, writing JML specifications for the Java modules of our toolsets has been and is routinely done. We are also applying our tools to themselves, thus providing non-trivial case studies demonstrating the practical utility of the tools.

ESC/Java2 and Loop have been applied to other case studies in the areas of internet voting [11], JavaCard applications [10], and web-based enterprise applications, for example. Some of these case studies are already part of our GForge [17]; the rest will be added shortly. We will be routinely re-executing these case studies as the tools evolve so as to validate the tools and ensure that their effectiveness is, in fact, improving.

6. Acknowledgments

This proposal is based upon the work of many people. Our collaborators are gratefully acknowledged on the first page as well as in the various sections of the proposal. We thank the anonymous referees for their helpful comments. This work is being supported by the Ireland Canada University Foundation as well as international grants (EU FP6) and national grants (from the Science Foundation Ireland and Enterprise Ireland).

References

[1] C.-B. Breunesse. *On JML: Topics in Tool-assisted Verification of Java Programs*. PhD thesis, Radboud University Nijmegen, 2005. in preparation.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[3] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Publishing Company, 2000.

[4] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, 1999.

[5] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In *Proceedings, CASSIS 2004*, Marseille, France, 2004. Elsevier Science, Inc. In press.

[6] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[8] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04*

(Boston, Massachusetts), volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, first edition, Aug. 1996.

[10] B. Jacobs. JavaCard program verification. In R. Boulton and P. Jackson, editors, *Theorem Proving in Higher Order Logics TPHOL 2001*, volume 2151 of *Lecture Notes in Computer Science*, pages 1–3. Springer–Verlag, 2001.

[11] B. Jacobs. Counting votes with formal methods. In C.Rattray, S. Majaraj, and C. Shankland, editors, *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, pages 241–257. Springer–Verlag, 2004.

[12] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*. Department of Computer Science, Iowa State University, 226 Atanasoff Hall, draft revision 1.94 edition, 2004.

[13] The MOBIUS project. http://mobius.inria.fr/.

[14] S. Ranise and D. Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *International Conference on Software Engineering and Formal Methods SEFM 2003*, Canberra, Australia, Sept. 2003. IEEE Computer Society.

[15] RTI: Health, Social, and Economics Research, Research Triangle Park, NC. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02-3, NIST, May 2002.

[16] SMT-LIB: The satisfiability modulo theories library. http://goedel.cs.uiowa.edu/smtlib/.

[17] The Systems Research Group GForge. http://sort.ucd.ie/.