Verification for legacy programs

Michael D. Ernst
MIT Computer Science & Artificial Intelligence Lab
mernst@csail.mit.edu

In the long run, programs should be written from the start with verification in mind. Programs written in such a way are likely to be much easier to verify. They will avoid hard-to-verify features, may have better designs, will be accompanied by full formal specifications, and may be annotated with verification information. However, even if programs *should* be written this way, not all of them will. In the short run, it is crucial to verify the legacy programs that make up our existing computing infrastructure, and to provide tools that assist programmers in performing verification tasks and — equally importantly — in shifting their mindset to one of program verification. I propose approaches to verification that may assist in reaching these goals.

The key idea underlying the approaches is specification inference (Section 1). This is a machine learning technique that produces, from an existing program, a (likely) specification of that program. Specifications are very frequently missing from real-world programs, but are required for verification. The inferred specification can serve as a goal for verification. I discuss three different approaches that can use such inferred specifications. One uses a heavyweight proof assistant (Section 2), one uses an automated theorem prover (Section 3, and one requires no user interaction but provides no guarantee (Section 4).

1 Inference of likely specifications

The verification techniques described in Sections 2–4 build upon ongoing work for obtaining an *operational abstraction* — a formal description of properties that held on a series of program runs and may be expected to hold on future runs. The task of generating an operational abstraction is also known as dynamic detection of likely invariants, or dynamic invariant detection.

Dynamic invariant detection is an important and practical problem. Operational abstractions have been used in verifying safety properties [VH98, NE02b, NE02c], automating theorem-proving [NE02a, NEG$^+$04], identifying refactoring opportunities [KEGN01], predicate abstraction [DDLE02, DLE03], generating test cases [XN03a, XN03b, Gup03, GH03], selecting and prioritizing test cases [HME03], explaining test failures [GV03], predicting incompatibilities in component upgrades [ME03, ME04a], error detection [RKS02, HL02, PRKR03, MP04, BE04], error isolation [XN02, LAZJ03], and choosing modalities [LE04], among other tasks. Dynamic invariant detection has been independently implemented by several research groups, and related tools that also produce formal descriptions of run-time behavior have seen wide use.

Dynamic detection of likely invariants [ECGN01] discovers likely invariants from program executions by instrumenting the target program (in source or binary form) to trace the variables of interest, running the instrumented program over a set of test cases, and postulating and checking invariants over values that the program computes. The essential idea is to use a generate-and-check algorithm to test a set of possible invariants against the observed values of the instrumented variables. (Each set of observed values at a program point is called a "sample".) The invariant detector reports those properties that are tested to a sufficient degree without falsification. The output includes properties such as "at entry to procedure `foo`, *myList* is sorted", "at exit from procedure `bar`, $return \geq myVar$" (where *return* stands for the return value), and "for all `Link` objects, $this.next.prev = this$". As with other dynamic approaches such as profiling, the accuracy of the results depends in part on the quality and completeness of the test cases. Even modest test suites produce good results in practice [NE02c, NE02b], and techniques exist for creating good test suites for invariant detection [HME03, GH03, XN03b]. In the remainder of this paper, for brevity we use "invariant" to mean "likely invariant", unless otherwise noted.

We have built an implementation named Daikon. The Daikon dynamic invariant detector is publicly available from `http://pag.csail.mit.edu/daikon`. Daikon operates on C, C++, Java, and Perl code, and on various other data formats. It produces output in a variety of outputs, including ones appropriate for those programming languages and for theorem provers such as LP [GG89], Isabelle [NPW02], and ESC/Java [FLL+02] (via its integration with JML [BCC+05]). It reports properties over both variables from the source program and also expressions composed of those variables. Its grammar of properties is relatively rich, permitting their use in a variety of realistic verification scenarios; but the grammar is also limited by the performance of the machine learning technique. Users can easily add new properties in order to make the grammar richer or to customize it for a particular application domain.

Implementing dynamic invariant detection efficiently is challenging, because of the great number of potential properties to check. We have recently developed new incremental algorithms that permit dynamic invariant detection to run over much larger data sets, and to use a much larger grammar, than previously possible [PE04]. Future work on dynamic invariant detection will continue to focus on performance.

2 Guiding human proofs

We propose to use execution to assist theorem-proving. Execution-based techniques such as testing can increase confidence in an implementation, provide intuition about behavior, and detect simple errors quickly. They cannot by themselves demonstrate correctness. However, they can aid theorem provers by suggesting necessary lemmas and providing tactics to structure proofs.

Theorem provers are powerful tools for ensuring that purported proofs are correct, that is, that proofs adhere to the rules of logic. The main hindrance to using theorem provers has been the amount of human input they require. General-purpose theorem provers for sufficiently powerful logics have acted less as automated verification tools than as interactive proof systems or proof assistants. Humans must provide them with two primary types of input: lemmas and tactics. Lemmas provide facts about the programs being verified, which are often necessary for correctness proofs. Tactics guide the prover in making choices during a proof, such as which lemmas to apply or whether to reason by cases or by induction.

The focus of much previous work on making provers easier to use has been on analyzing syntactic structures in axioms and conjectures in order to generate potentially useful lemmas and tactics. When these lemmas and tactics do not suffice, humans must provide additional input based on their understanding of the semantic content of the axioms and conjectures. Often this understanding is faulty or incomplete. The focus of the work described here is on making it easier to use theorem provers for verifying distributed algorithms by reducing the need for this kind of human input. To this end, we use a dynamic analysis of the results of executing a program, in addition to a static analysis of the program's text and of its test suite, to increase human insight, to discover semantic content in the program's behavior, and to generate potentially useful lemmas and tactics for correctness proofs.

This is a new use for execution, which has been a traditional part of algorithm and system development, but does not yet play a direct part in formal verification. Because execution requires little human effort, it has traditionally served as a powerful prelude to formal verification, a task that requires much greater human effort. When used for testing, execution can reveal departures from desired behavior that can be corrected before attempting to prove code correct. Execution can serve in additional ways as a prelude to formal verification. Tools for dynamic program analysis can extract descriptions of program behavior from executions, and programmers can match the extracted descriptions against their expectations. Unlike the traditional use of execution to test behavior, this use can reveal unexpected behaviors, not just departures from anticipated behaviors. Furthermore, simulated execution can be used to test specifications (expressed, for example, as abstract programs) in the same way that actual execution tests programs, even in the absence of a complete implementation.

These uses of execution to test programs and specifications occur before verification and are largely disjoint from it. We propose to integrate information obtained from execution into the process of formal verification.

First, we use descriptions of program behavior, extracted by dynamic analysis from executions, as lemmas in proofs. Unlike human proofs, which are peppered with phrases like "it is obvious that," machine-checked proofs often require many explicit lemmas. Some lemmas are tedious and obvious, some are not. In either case, using dynamic program analysis to provide these lemmas saves human effort.

Second, information used to construct test suites can also play a role in verification. During testing, such information ensures adequate test coverage by ensuring that all interesting cases—normal, abnormal, or borderline, as determined by the programmer, by the tester, or by static analysis—are tested. During verification, this information can be used to supply proof tactics, for example, to choose helpful case splits. Thus, tactics generated from test suites for simulated execution can complement tactics built into the prover in reducing human input.

We have applied these techniques to correctness proofs of three distributed algorithms: the Peterson mutual exclusion algorithm [Pet81], an algorithm for ensuring memory atomicity in the presence of distributed caches [BGL02], and the Paxos algorithm for achieving distributed consensus [Lam98]. We performed each proof using two rather different theorem-provers: LP and Isabelle. Considering multiple algorithms and multiple theorem-provers suggests that our technique is general. We found that the lemmas and tactics that were automatically suggested by our tools eliminated on the order of 90% of user interaction with the tools (100%, in one case), and in one case led to a shorter proof than we had previously been aware of [NEG+04, NEG02].

3 Automated theorem prover

The experience with theorem-proving described in Section 2 indicates that humans can effectively use the guidance provided by a dynamic (runtime) analysis tool to assist them in theorem-proving. We have performed a pair of complementary experiments to further investigate the efficacy of dynamic analysis. Both experiments use the ESC/Java [FLL+02] tool, an automated theorem-prover, as an automatic measure of the adequacy of a specification — whether produced by a tool or by a human.

The first experiment [NE02b] assesses how accurate the results of the dynamic analysis are. Producing specifications by dynamic (runtime) analysis of program executions is potentially unsound, because the analyzed executions may not fully characterize all possible executions of the program. Surprisingly, small test suites captured nearly all program behavior required by the ESC/Java static checker. ESC/Java guaranteed that the implementations satisfy the generated specifications, and ensured the absence of runtime exceptions. Measured against this verification task, the generated specifications scored over 90% both on precision, a measure of soundness, and on recall, a measure of completeness.

This is a positive result for testing, because it suggests that dynamic analyses can capture all semantic information of interest for certain applications. The experimental results demonstrate that a specific technique, dynamic invariant detection, is effective at generating consistent, sufficient specifications for use by a static checker. Finally, the research shows that combining static and dynamic analyses over program specifications has benefits for users of each technique, guaranteeing soundness of the dynamic analysis and lessening the annotation burden for users of the static analysis.

The second experiment [NE02c] artificially degraded the quality of the dynamic analysis via use of unreasonably small test suites. Then, we performed a human experiment with 41 experienced programmers, in which each was asked to verify a small program using ESC/Java. Some were given no assistance; some were given Daikon output (dynamically detected likely invariants) obtained from a good test suite; some were given Daikon output obtained from a bad test suite; and some were assisted by a static technique (Houdini [FL01]) designed by the authors of ESC/Java. Some sort of user assistance seems necessary: users find the annotation task tedious, difficult, and unrewarding, and therefore often refuse to perform it [FJL01]. In other words, users seem to view the cost of annotation as greater than the benefits of static checking. Thus, we considered how to reduce the annotation burden.

In brief, the statistically significant results suggest that both tools contribute to success, and neither harms users in a measurable way. Unsoundness did not hinder users: even very inaccurate dynamic analysis output produced from tiny test suites is better than no assistance. Additionally, Houdini helps users to express more properties in fewer annotations, and Daikon helps users express more true properties than strictly required, with no time penalty. However, users reported concerns with Houdini's speed and opaqueness and with Daikon's verbosity on poor test suites.

4 Predicting incompatible software component upgrades

Large software systems are usually made up of a number of components, not all produced by a single development team. Ensuring that separately designed components can cooperate is a major challenge in building a large system, and the difficulty is compounded when components change over time. System integrators would often like to know whether an updated version of a software component can be added to a system without disrupting its correct operation.

We have developed a new technique to assess whether replacing one or more components of a software system by purportedly compatible components will change the behavior of the system. The technique operates prior to integrating the new components into the system, permitting quicker and cheaper identification of problems. It takes into account the system's use of the components, because an upgrade may be desirable in one context but undesirable in another. No formal specifications are required, permitting detection of problems due to errors in the components or errors in the system. Both external and internal behaviors can be compared, enabling detection of problems that are not immediately reflected in the system's output.

The two key techniques that underly our method are describing observed behavior (via dynamic invariant detection) and comparing those behaviors via logical implication. Operational abstractions are more useful for our purposes than human-written formal specifications of desired behavior, because we want to find incompatibilities induced by the actual behavior of the software.

Because operational abstractions are simply collections of statements in a formal language, they can be mechanically compared to determine if one abstraction is logically implied by another. If our system, using an automatic theorem prover, can verify that the abstraction describing a component's desired behavior is a logical consequence of the abstraction of its observed behavior during testing, we have evidence that the component will indeed function correctly.

We have formulated a general model for the semantics of a multi-component system, based on a division into modules with certain behaviors and certain expectations of the behavior of other modules. This model encompasses a variety of component interactions, including components with state, access to shared variables, callbacks, and upgrades that require the simultaneous replacement of multiple components. From this model, we use an algorithm to construct logical relations which, if they hold over the operational abstractions of a set of components, indicate that each component's expectations will be satisfied when the system executes. These relations are then checked using the Simplify automatic theorem prover. We have supplemented the basic technique with several enhancements that make it more effective in practical applications: these deal with nonlocal data and nondeterministic behavior, and allow the technique to differentiate between preexisting innocuous incompatibilities from new ones, which are more dangerous.

We have applied the technique to real-world components: Perl modules from the Comprehensive Perl Archive Network (CPAN) [ME03], and the main Linux system library, as used by 48 applications [ME04a]. In these case studies, our implementation verifies the safety of some upgrades, while discovering incompatibilities in other upgrades that cause applications to malfunction. Our technique's comparisons can be performed efficiently, and its rate of false positive warnings is low.

In ongoing research [ME04b], we are focusing on giving a precise description of the multi-component upgrade condition in the context of a simplified formal model. The model idealizes the operational abstractions used by the technique to abstractions that soundly approximate the behavior of the components they describe. A soundness result for the formalization (which we have so far achieved for the case of a two-component upgrade) corresponds to a relative soundness of the real system, which restricts the source of unsoundness to the finiteness of the testing used in constructing the abstractions. Work on the formalization has also motivated refinements and led to a better understanding of the technique as implemented, including trade-offs between accuracy and performance.

5 Conclusion

All programmers test their programs, but research in verification has too often ignored the rich information that is available in test suites and in program executions. We propose to integrate such dynamic information into verification — and we have made significant steps toward such integration. Although dynamic analyses are inherently unsound, their results can nevertheless be of assistance to both humans and to automated tools, even when the goal is sound verification. Dynamic and static analyses have complementary strengths: dynamic analyses can often produce more, and more precise, properties, and static analyses can check these to determine which ones can be guaranteed. Furthermore, dynamic analyses may be an important stepping-stone in analyzing legacy programs and in educating programmers for the transition from current, error-prone methodologies to ones in which verification techniques and tools are commonplace.

References

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[BE04] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, pages 480–490, May 2004.

[BGL02] Andrej Bogdanov, Stephen J. Garland, and Nancy A. Lynch. Mechanical translation of I/O automaton specifications into first-order logic. In *FORTE*, November 2002.

[DDLE02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002. Draft. `http://pag.csail.mit.edu/~mernst/pubs/invariants-implications.ps`.

[DLE03] Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Lab for Computer Science, July 21, 2003.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, February 2001.

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2(4):97–108, February 2001.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, pages 500–517, Berlin, Germany, March 2001.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, June 2002.

[GG89] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch Prover. In *RTA-89*, pages 137–151, April 1989.

[GH03] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE*, pages 49–58, October 2003.

[Gup03] Neelam Gupta. Generating test data for dynamically discovering likely program invariants. In *WODA*, pages 21–24, May 2003.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *SPIN 2003*, pages 121–135, May 2003.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE*, pages 291–301, May 2002.

[HME03] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *ICSE*, pages 60–71, May 2003.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM*, pages 736–743, November 2001.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, June 2003.

[LE04] Lee Lin and Michael D. Ernst. Improving adaptability via program steering. In *ISSTA*, pages 206–216, July 2004.

[ME03] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, pages 287–296, September 2003.

[ME04a] Stephen McCamant and Michael D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, June 2004.

[ME04b] Stephen McCamant and Michael D. Ernst. Formalizing lightweight verification of software component composition. In *SAVCBS*, pages 47–54, October/November 2004.

[MP04] Leonardo Mariani and Mauro Pezzè. A technique for verifying component-based software. In *TACoS*, pages 17–30, March 2004.

[NE02a] Toh Ne Win and Michael D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Lab for Computer Science, May 25, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, pages 232–242, July 2002.

[NE02c] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *FSE*, pages 11–20, November 2002.

[NEG02] Toh Ne Win, Michael D. Ernst, and Stephen J. Garland. Connecting specifications, executions, and proofs: Reducing human interaction in theorem proving, October 2002.

[NEG+04] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *STTT*, 6(1):67–76, July 2004.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[PE04] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *FSE*, pages 23–32, November 2004.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[PRKR03] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG*, pages 273–276, September 2003.

[RKS02] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE*, pages 302–312, May 2002.

[VH98] Mandana Vaziri and Gerard Holzmann. Automatic detection of invariants in Spin. In *SPIN 1998*, November 1998.

[XN02] Tao Xie and David Notkin. Checking inside the black box: Regression fault exposure and localization based on value spectra differences. Technical Report UW-CSE-02-12-04, U. Wash. Dept. of Comp. Sci. & Eng., Seattle, WA, USA, December 2002.

[XN03a] Tao Xie and David Notkin. Exploiting synergy between testing and inferred partial specifications. In *WODA*, pages 17–20, May 2003.

[XN03b] Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. In *ASE*, pages 40–48, October 2003.