Model-Checking Software Using Precise Abstractions

Marsha Chechik and Arie Gurfinkel

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada
{chechik,arie}@cs.toronto.edu

1 Introduction

Formal verification techniques are not yet widely used in the software industry, perhaps because software tends to be more complex than hardware, and the penalty for bugs is often lower (software can be patched after the release). Instead, a large amount of time and money is being spent on software testing, which misses many subtle errors, especially in concurrent programs. Increased use of concurrency, e.g., due to the popularity of web services, and the surge of complex viruses which exploit security vulnerabilities of software, make the problem of creating a verifying compiler for production-quality code essential and urgent.

Many formal techniques can effectively analyze *models* of software. However, obtaining these models directly from the program text is non-trivial. Not only are there language issues (pointers, dynamic memory allocation, object-orientation, dynamic thread creation) to deal with, but the most important problem is automatically determining the level of abstraction for creating such a model: it should be simple enough to analyze and yet detailed enough to be conclusive on the properties of interest. Thus, we believe that part of the challenge of creating a verifying compiler is creation of fully automated light-weight verification and validation techniques which help detect bugs and yet scale to handle large complex software. Furthermore, the techniques should be supported by a methodology that enables developers to pose questions about correctness of their programs and effectively understand the results of the analysis.

Software model-checking [BPR01, VHB$^+$03] checks the program by either forcing it to "run" all of its behaviours up to a certain number of steps, or by static analysis. Pioneered by Microsoft's SLAM project, software model-checking has been successfully applied to checking Windows device drivers [BCLZ04], demonstrating that effective verification of single-threaded C programs is possible.

We believe that the existing techniques based on the CEGAR framework (see Figure 1) that use a theorem-prover for constructing models and a model-checker for exploration is a reasonable way to go for analyzing realistic programs. However, such methods should be enhanced in a number of directions if our goal is to create a truly effective program verifier. In Section 2, we describe our existing work on creating better abstractions. In Section 3, we discuss two approaches that we believe are essential for scaling automated analysis: reuse (via compositional analysis and regression verification) and combining static and dynamic reasoning. We conclude in Section 4.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 347–353, 2008.

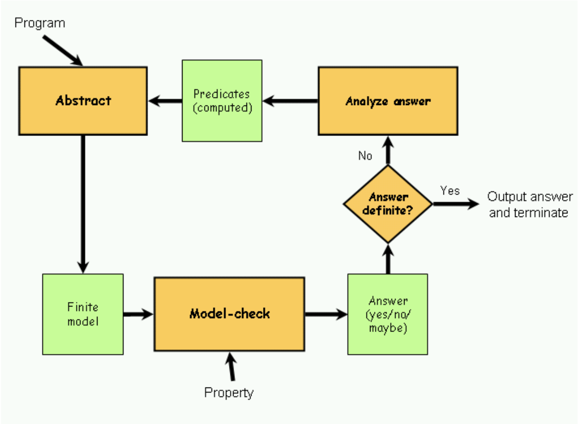

Fig. 1. Abstraction-refinement cycle

(a)

```
int x;
x = 0;
if (x > 0)
  {x++}
else
  {x--}
P1:
```

(b)

```
;
if (*)
  {;}
else
  {;}
P1:
```

Fig. 2. (a): a C program where line `P1` is not reachable; (b): an abstraction of (a) without predicates

2 Creating Precise Abstractions

Paraphrasing Reps and Sagiv [RSW04], we need abstractions that allow lack of precision and still enable reasoning. Thus, an abstraction is “good” if it can identify whether the analysis of a property of interest is conclusive (i.e., when the property is true/false in abstract model, it is true/false in the concrete one). With such an abstraction, we can trust all of the answers except those that explicitly indicate “no information”. Experience in static analysis [RSW04] showed that a logic with values other than just True and False can be used effectively to create abstractions that preserve both truth and falsity: when a property is inconclusive in the abstraction, the analysis simply returns a special value Maybe.

Traditional model-checking approaches already build such abstractions. For example, suppose we are interested in checking whether `P1` is reachable in a program in Figure 2(a). The initial abstraction for this program is just its control-flow graph, shown in Figure 2(b), and it is clear that `P1` is reachable by every path, i.e., regardless of the evaluation of the condition of the `if` statement. However, since the conditions is “unknown”, i.e., it is *neither* True nor False, it is not possible to convert it into a boolean model, expected by

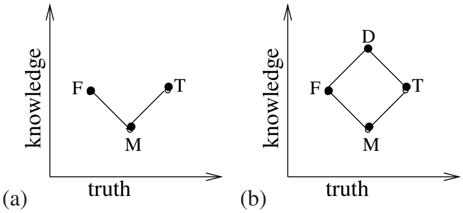

Fig. 3. Truth order vs. knowledge order of (a): Kleene and (b): Belnap logics

classical model-checking algorithms. Here, two choices are available. The standard approach, employed by such model-checkers as SLAM and BLAST, is to treat “unknown” as “non-deterministic”, i.e., as *either* True or False, which looses some of the information available in the abstraction, but allows the use of classical model-checking algorithms. An alternative approach is to extend the analysis to partial models that can represent “unknown” explicitly. In fact, several such alternatives, based on Kleene and Belnap logics (see Figure 3, where the additional values stand for “maybe” and “disagreement”) have been proposed over the years (e.g. [LT88, BG99, CDEG03, DGG97, GWC06]), but have not found their way into software model-checking practice because it is generally believed that they cannot be implemented efficiently. Thus, the additional power of already available rich abstractions remains untapped. Instead, the classical analysis needs to know exactly which execution of the `if` statement in the example in Figure 2(b) is feasible, which leads to generating an additional predicate and a refined abstraction. Clearly, this can and should be avoided.

To show that the additional reasoning power can be obtained without sacrificing performance, we have recently built a prototype software model-checker YASM [GC06] which implements the CEGAR framework using Belnap logic. The “disagreement” value is used to improve precision of our models. The remainder of this section comments on how our approach differs from the classical one, phase by phase.

The abstraction phase builds an abstract model, using a theorem-prover (YASM uses CVCLite [BB04]) to approximate the effect of each statement by a propositional formula over the available predicates. Abstracting each line of the program can take up to 2^n calls to the theorem prover, where n is the current number of predicates [GS97, BR01]. This number can be minimized by the application of constraint-satisfaction algorithms [Dec03], and we are currently experimenting with several CSP techniques to improve the performance of our tool. Overall, our abstraction phase is quite standard and is independent of our use of Belnap logic for the analysis.

For the model-checking phase, our current implementation uses a modification of our existing model-checker χChek [CDG02] which can naturally analyze models expressed in Belnap and Kleene logic. Just as other implementations of the CEGAR framework, we extended χChek's algorithms to exploit the control-flow graph of the program. We also discovered that it is relatively straightforward to incorporate and extend existing advances in software model-checking technology, such as interpolant-based predicate discovery [HJMM04] and lazy abstraction [HJMS02], into our framework.

Multi-valued witnesses and counterexamples produced by χChek correspond to partial proofs of correctness [GC03]. Thus, in the refinement phase, it is not necessary to check whether the execution produced by model-checking is feasible. Instead, we simply refine abstractions that produce the Maybe answer, by extracting new predicates for refinement from such partial proofs. Existing SLAM-like techniques can handle safety properties only (e.g., x is always positive, null pointer is never dereferenced, an assertion is always satisfied), partly because the traditional CEGAR framework depends on the linearity and the finiteness of generated counterexamples. Our approach can work regardless of the structure of the property, and thus can be applied to a larger set of properties, such as non-termination (to complement the technique of Cook et. al. [CPR05]) and liveness, as well as to programs with explicitly-defined *fair* computations.

The current version of the tool can check properties of C programs with complex language features such as structures, pointers, and recursion. We have also implemented lazy and eager abstraction and a limited form of non-determinism, which will lead to the analysis of concurrent programs. Our experience showed that the introduction of additional logic values does not reduce the feasibility of the analysis. In fact, the analysis remains as feasible as the classical one while making it possible to effectively reason about a larger class of properties.

3 Other Approaches to Effective Software Model-Checking

A verifying compiler should be able to effectively compile arbitrarily large and complex programs, from device drivers which can be effectively analyzed by state-of-the-art tools, to multi-user multi-threaded distributed systems. Thus, the problem of "scaling up" of software model-checking will remain pressing, and in this section we discuss two directions which we believe are essential for tackling this problem.

3.1 Reuse: Compositional and Regression Verification

Complexity of software is often addressed by decomposing a system into components. Thus, one way to improve scalability of a verification technique is by making it *compositional*. Over the years, various "assume-guarantee" techniques have been proposed to partition the verification effort across system components, and these can be adopted to reasoning about software. For example, to analyze a given thread, we can generate a most general environment that is sufficient to ensure that the desired property holds, and then check that the combination of other threads satisfies this environment. We also propose to apply symmetry reduction to build abstractions of concurrent systems with several similar components [WGC05]. We believe that a similar approach can be taken to facilitate the analysis of component-based and parameterized systems.

A major reason for SLAM's ability to analyze *recursive* programs is its compositional approach to processing functions. Instead of handling function calls by inlining, SLAM treats each function as a component, analyzes it once, and uses the computed *function summary* in the rest of the analysis.

However, the traditional notion of a component (i.e., a function, a module, a library, a thread), is not sufficient to effectively capture the complexity of software. This is especially highlighted by the recent popularity of non-traditional decomposition techniques such as aspect-based programming [EFB01].

An approach complementary to compositional verification is *regression verification*. As software gets changed frequently and needs to be "recompiled" (and thus reverified), the goal of this approach is to determine how much of the modified program needs to be reverified and reusing results of previous analyses whenever possible. In particular, it is necessary to determine when previous abstractions can be reused in checking the modified program. Further, changes in software often affect a large number of components, and we believe that the re-verification effort must be proportional to the amount of change, not to the number of affected components.

The lack of support for regression verification is particularly evident in the current applications of the CEGAR framework. In this framework, the abstraction of the program is constructed incrementally, with each new abstraction being a "small change" of the old. However, with the notable exception of [HJMS02, GC06], this is not taken into account during the verification phase. Early experience with regression verification shows a lot of promise. For example, Henzinger et al. [HJMS04] have demonstrated an almost "on-line" verification of some properties, where the program is verified as it is being written. Our own experience [GC06] shows that program analysis using Belnap logic that precisely identifies which results of a verification can be trusted, is particularly well suited to localizing the effort of regression verification. Moreover, our current work on merging [UC04] allows us to combine function and other component summaries obtained during different analysis passes through the program being modified.

3.2 Combining Static and Dynamic Approaches

The goal of static analysis is to establish a property of all executions of a program. In theory, it promises to completely eliminate dynamic analysis such as testing and runtime monitoring; moreover, this has been possible in practice, for analyzing relatively small components such as device drivers. We believe that static techniques alone may not scale to reasoning about large distributed programs that are routinely being built today, and may need to be supplemented by a careful application of dynamic analysis.

There are numerous ways in which static and dynamic analysis can be combined. For example, program execution can supplement theorem-proving in construction of abstract models [KGC04]: if the program reaches the desired state, then the value of the transition should be True; otherwise. no information is available.

Another approach is to combine results of testing to help navigate static analysis towards a possible error. We are currently exploring a variation of this approach by building an abstract model from regression test suites. Such suites are available with most large-scale software systems and can be thought of as detailed scenarios describing the system. We can combine these into partial behavioural models [UC04, WS00] to use in formal analysis.

4 Position

In this paper, we briefly discussed our position on using software model-checking for creating a verifying compiler. We are firm believers in automated symbolic verification which combines static and dynamic analyses, theorem-proving and model-checking, and think such techniques can be effectively extended to reasoning about complex software systems. We also believe that capturing the distinction between "the abstraction is not precise" and non-determinism is a key to pushing symbolic approaches towards reasoning about concurrency.

Currently, the automated verification research community has several competing approaches to analyzing software. We hope that bringing us together will facilitate tool sharing, so we can evaluate improvement of our tools against the state-of-the-art. We also hope that we can create and share a set of agreed-upon "requirements" for a verifying compiler. These requirements can take a form of a benchmark suite combining programs of varying complexity and size, with different language features and different correctness criteria.

References

[BB04] Barrett, C., Berezin, S.: CVC Lite: A New Implementation of the Cooperating Validity Checker. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 515–518. Springer, Heidelberg (2004)

[BCLZ04] Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: "Theorem Proving for Predicate Abstraction Refinement". In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, Springer, Heidelberg (2004)

[BG99] Bruns, G., Godefroid, P.: Model Checking Partial State Spaces with 3-Valued Temporal Logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)

[BPR01] Ball, T., Podelski, A., Rajamani, S.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)

[BR01] Ball, T., Rajamani, S.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)

[CDEG03] Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: "Multi-Valued Symbolic Model-Checking". ACM Transactions on Software Engineering and Methodology 12(4), 1–38 (2003)

[CDG02] Chechik, M., Devereux, B., Gurfinkel, A.: χChek: A Multi-Valued Model-Checker. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 505–509. Springer, Heidelberg (2002)

[CPR05] Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)

[Dec03] Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)

[DGG97] Dams, D., Gerth, R., Grumberg, O.: Abstract Interpretation of Reactive Systems. ACM Transactions on Programming Languages and Systems 2(19), 253–291 (1997)

[EFB01] Elrad, T., Filman, R., Bader, A.: Aspect-Oriented Programming: Introduction. Communications of the ACM, pp. 29–32 (October 2001)

[GC03] Gurfinkel, A., Chechik, M.: Proof-like Counterexamples. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 160–175. Springer, Heidelberg (2003)

[GC06] Gurfinkel, A., Chechik, M.: Why Waste a Perfectly Good Abstraction? In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)

[GS97] Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)

[GWC06] Gurfinkel, A., Wei, O., Chechik, M.: Systematic Construction of Abstractions for Model-Checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 381–397. Springer, Heidelberg (2005)

[HJMM04] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: Proceedings of 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), Venice, Italy, January 2004, pp. 232–244. ACM Press, New York (2004)

[HJMS02] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proceedings of 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), Portland, Oregon, January 2002, pp. 58–70. ACM Press, New York (2002)

[HJMS04] Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.: Extreme Model Checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)

[KGC04] Kroening, D., Groce, A., Clarke, E.: Counterexample Guided Abstraction Refinement via Program Execution. In: Proceedings of International Conference on Formal Engineering Methods, November 2004, pp. 224–238 (2004)

[LT88] Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proceedings of 3rd Annual Symposium on Logic in Computer Science (LICS 1988), pp. 203–210. IEEE Computer Society Press, Los Alamitos (1988)

[RSW04] Reps, T.W., Sagiv, M., Wilhelm, R.: Static Program Analysis via 3-Valued Logic. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 15–30. Springer, Heidelberg (2004)

[UC04] Uchitel, S., Chechik, M.: Merging Partial Behavioural Models. In: Proceedings of 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 2004, pp. 43–52 (2004)

[VHB$^+$03] Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Journal of Automated Software Engineering 10(2) (April 2003)

[WGC05] Wei, O., Gurfinkel, A., Chechik, M.: Identification and Counter Abstraction for Full Virtual Symmetry. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 285–300. Springer, Heidelberg (2005)

[WS00] Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. In: Proceedings of 22nd International Conference on Software Engineering (ICSE 2000), May 2000, pp. 314–323. ACM Press, New York (2000)