Early Results with Precision Abstraction: Using Data-flow Analysis to Improve the Scalability of Model Checking

Adam Brown James C. Browne Calvin Lin

Department of Computer Sciences
The University of Texas at Austin, Austin, TX 78712, USA,
{abrown,browne,lin}@cs.utexas.edu

Abstract

This paper presents a new state space reduction technique that applies to model checking of software. The new technique, precision abstraction*, borrows ideas from data-flow analysis to identify procedures that can be analyzed context-insensitively without affecting the accuracy of the verification of a given property. These context-insensitive procedures can then be represented with fewer states than would be needed context-sensitive analysis. Preliminary results indicate that the number of transitions in the analysis prescribed by our approach is at least 155 times fewer than the exhaustive analysis a model checker would otherwise perform.*

1 Introduction

Model checking is an important means of verifying properties of software. Model checking is computationally expensive because its cost depends on the number of states in the model being checked, which for complex systems can be quite large. Thus, previous work has explored ways to reduce the size of the model by abstracting away [5, 13] or simply removing states [1, 6, 7, 17, 23] that cannot affect the satisfiability of the property being checked. This paper introduces a novel method, which we call *precision abstraction*, for dramatically reducing the number of states in a model without sacrificing the accuracy of property verification.

The key observation is that current model checkers assume a fixed level of precision with respect to all procedure invocations, but this level of precision is not always needed. In particular, model checkers currently obey the semantics of procedure calls, effectively creating a model in which all procedure calls have been inlined. By contrast, the compiler community has long recognized that analysis can be performed at different levels of precision, so analysis time can often be reduced at the expense of precision. For example, an analysis that obeys the semantics of procedures calls is referred to as a *context-sensitive* analysis, while an analysis that blurs the distinction among different call sites and blurs the distinction among different calling contexts is known as a *context-insensitive* analysis.

Recently, the notion of adaptive analysis has been introduced [9]. Rather than analyzing every procedure with the same precision, a fast analysis is performed to identify those procedures that must be analyzed context-sensitively to produce an accurate result. All other procedures can be analyzed context-insensitively. Because typically only a handful of procedures require context-sensitive analysis [9], this adaptive analysis is extremely efficient. In this paper, we use the notion of adaptive analysis to identify those procedures that can be analyzed context-insensitively without affecting the accuracy of the verification of the given property. This information is then be used to construct a model that has significantly fewer states than one that uses full context-sensitivity.

This paper presents preliminary results that this idea, which we refer to as *Precision Abstraction*, can significantly reduce the number of states required for model checking. This work is part of a larger effort to unify model checking and data-flow analysis techniques to produce precise yet scalable verification tools. In some cases, data-flow analysis can be used to prove that a particular property holds for a given program, so more expensive model checking can be forsaken completely. In other cases, as described above, data-flow analysis can be performed to greatly reduce the cost of the subsequent model checking effort. Because our work attempts to fuse ideas from the model checking and static analysis communities, we also propose an empirical evaluation methodology that reports state space size, actual analysis time, and a measure of accuracy.

1-4244-0910-1/07/$20.00

2 Related Work

Existing solutions for state space reduction broadly fall into three categories: control abstraction, data abstraction, and unnecessary state removal. Control abstraction removes states by abstracting the control flow in the original program. Data abstraction removes states by abstracting the values that variables can take. Finally, unnecessary state removal techniques remove states which cannot affect the property being checked.

Control abstraction

Control abstraction techniques recognize that not all control flow in the original program affect the satisfiability of the property being checked. Partial order reduction (POR) [18], and by extension static partial order reduction (SPOR) [15], are primarily used when analyzing asynchronous systems. These techniques reduce the state space by combining states whose only difference is the relative ordering of changes. For example, assume that multiple processes in an asynchronous system may modify "local" variables, that is variables only visible to one process, and those variables do not affect the satisfiability of the property being checked. Then it suffices to only check one path through the state space containing all of the changes to those variables, instead of needing to examine all possible ordering of those updates. A primary difference between POR and SPOR is that the former is done during the execution of a model checker and the latter is performed when generating the input to a model checker and thus requires no changes to an existing model checker.

Abstraction refinement techniques, such as CEGAR [5] and lazy abstraction [13], assume an overly simplified model. Then, as the system finds potential errors, it incrementally refines the model by adding back in states and transitions of the more complete model. Predicate abstraction [2, 8] generates an abstracted model given a finite set of predicates over the variables in the model. Within the context of data-flow analysis, these techniques are comparable to *path-sensitive analysis*, wherein the path taken to a particular statement is considered when analyzing the statement.

Existing control abstraction techniques fit into a larger category that we will call *precision abstraction*. With precision abstraction, we aim to identify portions of the program/model that do not require the full precision of flow- and context-sensitivity. In general, existing control abstraction techniques only focus on flow-sensitivity, in that they operate on a fully context-sensitive representation of a program. We discuss our proposal for precision abstraction in Section 3.

Data abstraction

Data abstraction techniques reduce the state space by restricting or abstracting the data values that variables in the program may assume. For example, it may suffice for a loop count variable to be constrained modulo some integer, or a variable's value may only matter based on its sign (positive, negative, or zero). Similarly, the contents of a queue may only be significant if they contain an element of a certain type and all other values are equivalent and indistinguishable.

In general, data abstraction techniques are not limited by the property being examined. However, these techniques are not easily automated. It is possible that the approximation generated by the data abstractions obscures the validity of the model checking result.

Unnecessary state removal

Unnecessary state removal techniques include slicing [7, 17, 23] and cone of influence (COI) [1, 6] techniques. These techniques rely on the insight that states that do not affect state mentioned in the specification cannot affect the validity of the property. Slicing and COI primarily differ in the representation to which they are applied. Slicing typically occurs on a description of the model (for example, Promela), whereas COI is performed on the actual model, even as it is being generated in an on-the-fly model construction. The literature often uses "slicing" in place of "cone of influence".

Both of these techniques suffer from two main shortcomings. These shortcomings arise because the property being checked may contain several atomic propositions. First, slicing and COI must retain any state which affects any of the atomic propositions without considering the other atomic propositions. If that state does not belong to any path affecting the other atomic propositions, then it will have been needlessly included in the model.

Adaptive Analysis

Iterative flow analysis [19] is an analysis technique that adjusts its precision automatically in response to the quality of the results. Plevyak and Chien use this algorithm to determine the concrete types of objects in programs written using the Concurrent Aggregates object-oriented language.

More recently, Guyer and Lin [9] describe an adaptive pointer analysis algorithm—the Client Driven algorithm—that generalizes this approach to apply to typestate problems [22], which include the type of flow-sensitive problems that are typically necessary for verifying deep properties about software. In addition, Guyer and Lin's algorithm can provides adaptivity to the pointer analysis, which

is often the most costly aspect of static analysis. Our work uses the initial phase of Guyer and Lin's Client Driven algorithm to identify procedures that should be analyzed context-insensitively. We then use this information in model construction.

3 Precision Abstraction

Analogous to the client-driven analysis technique, our notion of precision abstraction first identifies the portions of a program that require context-sensitivity based on the needs of the property to be model checked. Our compiler then generates a model that matches the determined precision policy. Finally, this property-specific model is model checked using an existing state-of-the-art model checker.

In our compiler, the property to be verified is specified using a simple annotation language [11] that defines a fairly general class of typestate problems. There is a straightforward translation from these property specifications to Linear Time Logic [6].

Implementation

We have implemented precision analysis using the Broadway and C-Breeze compiler infrastructures [10, 16]. As such, we are currently limited to those properties that can be expressed as a typestate problem [22]. We use the first pass of the client-driven analysis in Broadway to identify the procedures that require context-sensitive analysis.

Then, we use this information to generate a SPIN [14] model that encodes the context-sensitivity information. Because Promela, SPIN's modeling language, does not provide a way to represent this information directly, we must generate a model that encodes our intentions. At the simplest level, procedure calls to context-insensitive procedures are turned into `goto`s to the single copy of code representing the procedure.

For context-sensitive procedures, we include a copy of the procedure's code at each callsite. In this way, we emulate inlining, which is commonly used to perform context-sensitive analysis.

However, this approach of inlining procedures presents a problem with context-sensitive, recursive (or mutually-recursive) procedures. To deal with these procedures, we employ the technique of creating a new Promela `proctype` for the recursive procedure and using a local `chan` to return values. In this way, we can represent the recursive procedure call by running the generated `proctype`, and SPIN will analyze each invocation of the procedure's `proctype` separately, achieving context-sensitivity. With few exceptions [2], this work is in contrast to most existing model extractors for software which do not support recursion [7, 3, 4, 12].

	CI	CDA	CS	CS / CDA
blackhole	959	1239	>219999	>177.56
bind	311	551	>379999	>689.65
pfinger	43	43	7003	162.86
muh205	41	41	8930	217.80

Table 1. Comparison of Broadway procedure locations

4 Evaluation

4.1 Methodology

Our work combines ideas from both data-flow analysis and model checking. Thus, it requires comparison to work in both of these areas. Unfortunately, these two areas have different evaluation criteria. The data-flow analysis community is concerned with runtime characteristics and number of potential errors (violations of the specification) identified. However, the model checking community evaluates a solution based on the number of states in the generated model used to verify that there are no violations of a property. Additionally, most model checkers terminate once they have identified the first violation of a property.

The model checking community's choice to present state space size as their primary metric prevents the ability to make qualitative comparisons. As with any exhaustive search technique, the cost of the search heuristic affects the overall performance. In order to make informed comparisons between state space reduction techniques, the model checking community needs to report empirical results that include running time, both total running time and the overhead of the individual heuristic or state space reduction technique.

In general, we feel that the model checking community should adopt a methodology in which all violations are identified and reported. Within the context of checking software, simply identifying the first error places a heavy burden on the user to continuously re-run the analysis after each individual violation has been identified and resolved.

4.2 Preliminary Results

We are currently finishing our implementation of precision abstraction. Thus, we cannot provide direct comparison of the running time and state space. However, using the Broadway compiler, we have obtained a measure of the size of the resulting models.

We first present a measure of the number of procedure contexts that are analyzed with three different precision policies: context-insensitive (CI), context sensitive (CS), and client-driven analysis (CDA). In Broadway parlance,

	CI	CDA	CS	CS / CDA
blackhole	3865	5265	>819997	>155.74
bind	1273	2061	>1449996	>703.54
pfinger	150	150	24361	162.40
muh205	157	157	30114	191.80

Table 2. Comparison of Broadway statement locations

every time a procedure is analyzed in a new context, it is referred to as a *procedure location*. Table 1 presents this information for a sample of analyses present in Broadway. Guyer's thesis on Broadway presents descriptions of the programs and analyses used [10].

Analogous to procedure locations, in Broadway parlance, a statement location represents a statement in the original program, along with a calling context. Table 2 presents a similar result for statement locations for the same set of programs and analyses. The number of statement locations corresponds to the number of transitions in the resulting model.

5 Conclusions

Precision abstraction provides a promising approach to combining model checking and data-flow analysis. Using adaptive analysis, we have preliminary results that indicate that the number of transitions in a software program's analysis can be reduced by three orders of magnitude. It still remains to show how this reduction relates to the number of states in the resulting model and to the cost of model checking.

Context-sensitivity presents just one possible approach to reducing the precision of an analysis without impacting the accuracy of the result. We plan to extend this work to support *flow-insensitive* analysis, which does not respect the order in which program instructions are executed.

In general, the model checking and data-flow analysis communities are increasingly finding common ground in the types of software properties that they wish to verify. Unfortunately, while the theoretical similarities between the two approaches has been recognized [20, 21], there has been too little cross-pollinization of ideas between the two communities. Precision abstraction presents our first contribution in combining the best of model checking techniques and static analysis techniques.

Acknowledgments. This work is supported by the National Science Foundation under grant # CNS-0509354 ("Collaborative Research: CSR-AES: Unification of Verification and Validation Methods") and grant # ACI-0313263 ("ITR/SW: Compiler Techniques for Improving Software Quality").

References

[1] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 29–40, London, UK, 1993. Springer-Verlag.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, 2001.

[3] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. *IEEE Trans. Softw. Eng.*, 30:388–402, 2004.

[4] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169, 2000.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, 2000.

[8] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[9] S. Guyer and C. Lin. Client-driven pointer analysis. In *In International Static Analysis Symposium*, 2003.

[10] S. Z. Guyer. *Incorporating Domain-Specific Information into the Compilation Process*. PhD thesis, The University of Texas at Austin, 2003.

[11] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 39–52, New York, NY, USA, 1999. ACM Press.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN), Lecture Notes in Computer Science 2648*, pages 235–239. Springer, 2003.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[14] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[15] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static partial order reduction. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357. Springer-Verlag, 1998.

[16] C. Lin, S. Z. Guyer, and D. Jimenez. The c-breeze compiler infrastructure. Technical Report TR-01-43, The University of Texas at Austin, November 2001.

[17] L. Millett and T. Teitelbaum. Slicing promela and its applications to protocol understanding and analysis. In *4th International SPIN Workshop*, 1998.

[18] D. Peled. Ten years of partial order reduction. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28. Springer-Verlag, 1998.

[19] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340, New York, NY, USA, 1994. ACM Press.

[20] D. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM Symp. Principles of Programming Languages, San Diego*, 1998.

[21] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In G. Levi, editor, *Proc. 5th Static Analysis Symposium*, volume 1503 of *Lecture Notes on Computer Science*. Springer, September 1998.

[22] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[23] S. Vasudevan and J. Abraham. Static program transformation for efficient software model checking. In *WCC 2004*, 2004.