# SATABS: SAT-based Predicate Abstraction for ANSI-C*

Edmund Clarke[1] Daniel Kroening[2] Natasha Sharygina[1,3] Karen Yorav[4]

[1] Carnegie Mellon University, School of Computer Science
[2] ETH Zuerich, Switzerland
[3] Carnegie Mellon University, Software Engineering Institute
[4] IBM, Haifa, Israel

**Abstract.** This paper presents a model checking tool, SatAbs, that implements a predicate abstraction refinement loop. Existing software verification tools such as Slam, Blast, or Magic use decision procedures for abstraction and simulation that are limited to integers. SatAbs overcomes these limitations by using a SAT-solver. This allows the model checker to handle the semantics of the ANSI-C standard accurately. This includes a sound treatment of bit-vector overflow, and of the ANSI-C pointer arithmetic constructs.

## 1 Introduction

In the hardware domain, Model Checking [1] has become a well-established formal verification technique. In contrast to that, the software industry mostly relies on non-exhaustive techniques such as testing.

There are two issues that prohibit a wide-spread use of model checking tools for commercial software. First of all, most model checking tools do not scale gracefully when applied to software of substantial size. Thus, much of the research on model checking has focused on improving scalability. The second issue is that most model checking tools that are available use input languages that are not used for programming. The software system has to be translated into the input language of the model checker.

SatAbs, the tool presented in this paper, is geared towards application by software engineers. ANSI-C is one of the most popular programming languages, in particular for safety critical embedded software. Thus, the tool was designed

* This research was sponsored by the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU and was conducted as part of the PACC project at the CMU Software Engineering Institute (SEI). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

to take ANSI-C programs as input. In SatAbs, a special emphasis was made on supporting a rich subset of the ANSI-C language. The bit-vector semantics is modeled accurately, and thus the tool is able to detect errors that are related to bit-level operators and arithmetic overflow.

In order to address the scalability problem, SatAbs automatically computes an abstraction of the program given as input. Abstraction is one principal method in state space reduction of software systems. *Predicate abstraction* [2, 3] is one of the most popular and widely applied methods. It abstracts data by only keeping track of certain predicates. Each predicate is represented by a Boolean variable in the abstract model, while the original variables are eliminated. The abstract program is created using *Existential Abstraction* [4], which is a conservative abstraction for reachability properties. If the property holds on the abstract model, it also holds on the original program.

The drawback of the conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to a concrete counterexample. This is called a *spurious counterexample*. When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of predicates in a way that eliminates this counterexample. This is automated by *Counterexample Guided Abstraction Refinement* [5–7].

*Related Work.* Counterexample guided abstraction refinement for ANSI-C programs was promoted by the success of the Slam project at Microsoft [6]. Thus, there are already a number of other implementations, such as Magic [8], ComFoRT [9] and Blast [**?**]. All these projects have support for concurrent software. Both Slam and Blast now implement forms of lazy abstraction.

The feature that distinguishes SatAbs from these existing tools is the tight integration of a SAT solver into the abstraction, simulation, and refinement steps of the abstraction refinement loop. This allows precise encodings of the semantics of the ANSI-C language, including pointer-arithmetic and bit-vector overflow. In contrast to that, all the tools mentioned above rely on external theorem provers to reason about the programming language constructs. Initially, Slam, Blast, Magic, and ComFoRT used the theorem prover Simplify [10], which supports linear arithmetic on real numbers only. The remaining operators are approximated by means of uninterpreted functions. The Slam project replaced Simplify by Zapato [11], which provides better performance, but no support for bit-vectors.

The use of propositional logic and a SAT solver to reason about ANSI-C language constructs is already found in CBMC [12], a Bounded Model Checker for ANSI-C programs. There is a prototype of Slam, which has been integrated with parts of CBMC to reason about bit-vector constructs [13]. This version has found a previously unknown bug in Windows.

In [14], Lahiri, Bryant, and Cook use the SAT-based quantification engine implemented in SatAbs in order to compute abstractions of C programs. However, the algorithm uses unbounded integer semantics for the program variables, and does not support bit-vector operators. An integration into a full abstraction refinement loop is not reported.

In order to make SatAbs applicable to a wide range of low-level programs, SatAbs supports most constructs found in the ANSI-C language. In particular, it has support for arrays (with possibly unbounded size), and unions. None of the tools cited above provides these features. SatAbs is integrated into the graphical user interface (GUI) of CBMC. The user interface allows the user to step through counterexample traces generated by SatAbs as if using a debugger. The GUI is described in more detail in [12].

## 2 Using SAT for Predicate Abstraction and Refinement

This section provides a short overview of the algorithm implemented by SatAbs. For more information on the algorithm in the case of sequential code, we refer the reader to [15]. The algorithm is extended to concurrent programs with asynchronous interleaving semantics in [16].

SatAbs uses SAT-based Boolean quantification in order to compute the abstract model. Let $S$ denote the set of concrete states, $R$ the concrete transition relation, and $\alpha(x)$ with $x \in S$ the abstraction function. The abstract model can make a transition from an abstract state $\hat{x}$ to $\hat{x}'$ iff there is a transition from $x$ to $x'$ in the concrete model and $x$ is abstracted to $\hat{x}$ and $x'$ is abstracted to $\hat{x}'$. Formally, the abstract transition relation is denoted by $\hat{R}$.

$$\hat{R} := \{(\hat{x}, \hat{x}') \mid \exists x, x' \in S : R(x, x') \wedge \alpha(x) = \hat{x} \wedge \alpha(x') = \hat{x}'\}$$

This formula is transformed into CNF by replacing the bit-vector arithmetic operators by arithmetic circuits. Due to the quantification over the abstract states this corresponds to an all-SAT instance. For efficiency, SatAbs overapproximates $\hat{R}$ by partitioning the predicates into clusters [17].

The abstract model is passed to a model checker. SatAbs support a variety of model checkers, e.g., Moped, SPIN, NuSMV, and a QBF-based symbolic simulator. If the model checker returns a counterexample, it has to be simulated on the original code to check if it is spurious.

Given an abstract error trace, SatAbs first checks if it contains any spurious transitions. These spurious transitions are caused by the partitioning done during the computation of the abstraction. SatAbs forms a SAT instance for each transition in the error trace. If it is found to be unsatisfiable, the transition is spurious. As described in [17], the tool uses the unsatisfiable core of the instance for efficient refinement.

The absence of individual spurious transitions does not guarantee that the error trace is real. Thus, SatAbs forms another SAT instance. It corresponds to Bounded Model Checking on the original program following the control flow given by the abstract error trace. If satisfiable, SatAbs builds an error trace from the satisfying assignment, which shows the path to the error. If unsatisfiable, the abstract model is refined by adding predicates using weakest preconditions.

SatAbs was applied to system-level descriptions given in SpecC, a concurrent variant of ANSI-C [16]. Also, in [18], SatAbs was used for equivalence checking: SatAbs verified weak bi-simulation of an ANSI-C program and a circuit given in Verilog.

## 3 Conclusion

This paper presents an implementation of previously presented techniques for verifying ANSI-C programs. The contribution of SatAbs is its emphasis on precise encodings of the programming language constructs. The tool supports one of the most popular programming languages, ANSI-C. In contrast to other tools, it supports language features such as bit-vector operators, arrays, and unions. It comes with a graphical user interface that resembles a debugger. The distinguishing feature of SatAbs is the tight integration with a SAT solver.

## References


1. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
2. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254 of LNCS., Springer (1997) 72–83
3. Colón, M., Uribe, T.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV. Volume 1427 of LNCS., Springer (1998) 293–304
4. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. In: Principles of Programming Languages. (1992)
5. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1995)
6. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
8. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. In: ICSE. (2003) 385–395
9. Ivers, J., Sharygina, N.: Overview of ComFoRT: A Model Checking Reasoning Framework. Technical Report CMU/SEI-2004-TN-018, CMU SEI (2004)
10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
11. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: CAV. (2004)
12. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Volume 2988 of LNCS., Springer (2004) 168–176
13. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate Theorem Proving for Program Analysis. Technical Report 464, ETH Zurich, Computer Science (2004)
14. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: CAV. (2003) 141–153
15. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design **25** (2004) 105–127
16. Jain, H., Clarke, E., Kroening, D.: Verification of SpecC and Verilog using predicate abstraction. In: Proceedings of MEMOCODE 2004, IEEE (2004) 7–16
17. Clarke, E., Jain, H., Kroening, D.: Predicate Abstraction and Refinement Techniques for Verifying Verilog. Technical Report CMU-CS-04-139 (2004)
18. Kroening, D., Clarke, E.: Checking consistency of C and Verilog using predicate abstraction and induction. In: Proceedings of ICCAD, IEEE (2004) 66–72