The Slam Project: Debugging System Software via Static Analysis[*]

Thomas Ball and Sriram K. Rajamani
{tball,sriram}@microsoft.com
Microsoft Research
http://research.microsoft.com/slam/

Abstract. The goal of the Slam project is to check whether or not a program obeys "API usage rules" that specify what it means to be a good client of an API. The Slam toolkit statically analyzes a C program to determine whether or not it violates given usage rules. The toolkit has two unique aspects: it does not require the programmer to annotate the source program (invariants are inferred); it minimizes noise (false error messages) through a process known as "counterexample-driven refinement". Slam exploits and extends results from program analysis, model checking and automated deduction. We have successfully applied the Slam toolkit to Windows XP device drivers, to both validate behavior and find defects in their usage of kernel APIs.

Context. Today, many programmers are realizing the benefits of using languages with static type systems. By providing simple specifications about the form of program data, programmers receive useful compile-time error messages or guarantees about the behavior of their (type-correct) programs. Getting additional checking beyond the confines of a particular type system generally requires programmers to use assertions and perform testing. A number of projects have started to focus on statically checking programs against user-supplied specifications, using techniques from program analysis [18, 19], model checking [21, 17, 22], and automated deduction [16, 12].

Specification. The goal of the Slam project is to check temporal safety properties of sequential C programs [7]. Roughly stated, temporal safety properties are those properties whose violation is witnessed by a finite execution trace (see [24] for a formal definition). A simple example of a safety property is that a lock should be alternatingly acquired and released. We encode temporal safety properties in a language called Slic (Specification Language for Interface Checking) [9], which allows the definition of a safety automaton [30, 29] that monitors the execution behavior of a program at the level of function calls and returns. The automaton can read (but not modify) the state of the C program that is visible at the function call/return interface, maintain a history, and signal when a bad state occurs. We have developed Slic specifications for a variety of Windows XP driver properties, ranging from simple locking properties (such as given above) to complex properties dealing with completion routines, plug-and-play, and power management.

Given a program P and a Slic specification S, a preprocessor creates an instrumented program P' such that a unique label `ERROR` is reachable in P' if-and-only-if P does not satisfy S. The goal then shifts to determining whether or not the `ERROR` label is reachable in P', a generally undecidable problem.

Design. The basic design of the Slam process is to iterate the creation, analysis and refinement of program abstractions, until either a feasible execution path in P' to `ERROR` is found, the program P' is validated (`ERROR` is shown not to be reachable), or we run out of resources or patience.

The Slam process creates a sound *boolean program* abstraction B' of the C program P'.[1] Boolean programs have all the control-flow constructs of C programs, but contain only boolean variables. Each boolean variable in B' conservatively tracks the state of a predicate (boolean expression) in the C program. Boolean programs are created automatically using the technique of *predicate abstraction* [20]. If a *reachability* analysis of B' determines that the label `ERROR` is not reachable in B' then it is not reachable in P'. It is possible that B' may be too coarse an abstraction of P' (that is, `ERROR` is reachable in B' via a path p but `ERROR` is not reachable in P' via p). We apply a method known as *counterexample-driven refinement* [23, 28, 27] to create a more precise boolean program (by adding new predicates/boolean variables) that does not contain the spurious path p (or other paths that are spurious for the same reason p is). Termination of the Slam process is addressed below.

We expect the Slam process to work well for programs whose behavior is governed by an underlying finite state protocol. Seen in this light, the goal of Slam is to tease out the underlying "protocol" state machine from the code, to a level of precision that is good enough to find real errors

---

[*] Presented by the first author.

POPL '02, Jan. 16-18, 2002 Portland, OR USA

[1] Of course, whenever one hears a claim that an analysis of C code is "sound", one must ask "sound with respect to what assumptions?" because of the (potential) use of arbitrary pointer arithmetic. In Slam, we guarantee soundness under the assumption that the C program obeys a "logical memory model' in which the expressions `*p` and `*(p+i)` refer to the same object. Another analysis (see the work on CCured presented at this symposium [25]) is needed to discharge the "logical memory" assumption.

or validate the code. For example, while a video card driver may have a huge data path, most of this data has no bearing on the driver's interaction with the operating system. However, some of the driver data definitely are relevant to this interaction, and correlations between these data may need to be tracked.

Implementation. Three basic tools comprise the SLAM toolkit (in addition to the SLIC preprocessor):

- C2BP, a tool that transforms a C program P into a boolean program $\mathcal{BP}(P, E)$ with respect to a set of predicates E [2, 3]. C2BP translates each procedure of the C program separately, enabling it to scale to large programs. Using the theory of abstract interpretation [13], we have characterized the precision of the boolean program abstractions created by C2BP [4].
- BEBOP, a tool for performing reachability analysis of boolean programs [6, 8]. BEBOP combines interprocedural dataflow analysis in the style of [26] with Binary Decision Diagrams [10, 11] (BDDs) to efficiently represent the reachable states of the boolean program at each program point.
- NEWTON, a tool that discovers additional predicates to refine the boolean program, by analyzing the feasibility of paths in the C program.

The SLAM process starts with an initial set of predicates E_0 derived from the SLIC specification, and iterates the following steps:

1. Apply C2BP to construct the boolean program $\mathcal{BP}(P', E_i)$.
2. Apply BEBOP to check if there is a path p_i in $\mathcal{BP}(P', E_i)$ that reaches the `ERROR` label. If BEBOP determines that `ERROR` is not reachable, then P satisfies the SLIC specification and the process terminates.
3. If there is such a path p_i, then NEWTON checks if p_i is feasible in P'. There are three possible outcomes: "yes", the process terminates with an error path p_i; "no", in which case NEWTON finds a set of predicates F_i that "explain" the infeasibility of path p_i in P'; "maybe", the incompleteness of the underlying theorem prover may cause this outcome, in which case user input is required.
4. Let $E_{i+1} := E_i \cup F_i$, and $i := i + 1$, and proceed to the next iteration.

Termination: Theory and Practice. We have proved a strong relationship between a process based on iterative refinement of abstractions (such as in SLAM) and traditional fixpoint analyses with widening (which is used to ensure the termination of abstract interpretations in domains with infinite ascending chains) [5]. Using widening, the latter process always will terminate, but it may not give a definite result ("error found" or "program validated"). We have shown that if there is an oracle that can provide a "widening schedule" that causes the latter method to terminate with a definite result then an iterative refinement process (which does not rely on an oracle) will terminate with a definite result. Intuitively, this means that iterative refinement has the effect of exploring the entire state space of all possible sequences of widenings.

In practice, the SLAM process has terminated for all drivers within 20 iterations. Our major concern has been with the overall running time of the process. So far, we are able to analyze programs on the order of 10,000 lines, and abstractions with several hundred boolean variables in the range of minutes to a half hour. In practice, we find that most of the predicates SLAM generates are simple equalities with possible pointer dereferences. For this class of predicates, we believe it is possible to scale the SLAM process to several 100,000 lines of code through optimizations outlined below.

A major expense in the SLAM process is the reachability step (BEBOP), which has worst case running time $O(N(GL)^3)$, where N is the size of the boolean program, G is the number of global states, and L is the maximum number of local states over all procedures. The number of states is exponential (in the worst-case) in the maximal number of variables in scope.

The key to scaling for the SLAM process is in controlling the complexity of the boolean program abstraction. Satyaki Das has implemented a predicate abstraction technique based on successive approximations [15] in the SLAM toolkit, which has proven quite useful in this regard. Also relevant here is the paper on "lazy abstraction" in this symposium [21].

Additionally, there is substantial overhead in having to iterate the SLAM process many times, which can be addressed by both the "lazy abstraction" method as well as methods for heuristically determining a "good" initial set of predicates. Westley Weimer has implemented an algorithm in SLAM that, given the set of predicates present in the SLIC specification, determines what other predicates in the C program will very likely be needed in the future. This technique, based on the value-flow graph [14], greatly reduces the number of iterations of the SLAM process.

Challenges. We summarize by discussing some of the challenges inherent in the endeavor of checking user-supplied properties of software.

Specification burden. The creation of correct specifications is a hard problem requiring human time and energy (in the extreme, it is as hard as writing a program). If the effort put into developing specifications is not paid back in terms of discovered defects, then there is little incentive to develop specifications in the first place. We focused our specification effort at the level of the API so that specifications may be reused across different programs using the API. SLIC specifications can be partial. We started by first specifying a small set of errors in SLIC, and then gradually enlarging the set. Nevertheless, the complexity of the device driver API meant that it took considerable effort to arrive at a useful specification that found real defects. The "chicken and egg" problem of specifications is the topic of a paper in this symposium [1].

Annotation burden. By "annotation", we mean a modification to the program text inserted by a programmer explicitly to help an analysis tool make progress. Examples of such annotations include loop invariants and pre- and post-conditions for procedures, such as required by the ESC-Java tool [16]. In SLAM, annotations are not required. Instead, the abstract fixpoint analysis of the BEBOP tool discovers inductive invariants (loop invariants as well as procedure call summaries) expressed as a boolean combination of the predicates that are input to the C2BP tool.

Output. Generating good explanations of errors and their causes is a complicated affair, made more difficult as the expressivity of the specification language increases. When the SLAM toolkit finds an error, it presents it as an error path in the source code using an interface that resembles a source level debugger. However, there is sometimes an overwhelming amount of detail in these traces. We are developing techniques for presenting both short and detailed summaries of errors.

Soundness/Completeness/Usefulness. An analysis is "sound" if every true error is reported by the analysis, "complete" if every reported error is a true error (no noise), and "useful" if it finds errors that someone (programmers, testers, customers) cares about. Defect detection tools such as LCLint [19], Metal [18] and PREfix [12] are neither sound nor complete, yet are demonstrably useful. SLAM is sound (relative to the assumptions stated before), incomplete and is starting to demonstrate usefulness in the domain of device drivers.

Acknowledgements. Many people have contributed to the SLAM project. We have been fortunate to have many excellent interns who helped push the project forward over the summer months of 2000 and 2001. Sagar Chaki, Rupak Majumdar and Todd Millstein were 2000 summer interns. Sagar Chaki, Satyaki Das, Robby and Westley Weimer were 2001 summer interns. We have had a long and fruitful collaboration with Andreas Podelski, who has helped us understand the theoretical limits of the SLAM approach.

The SLAM toolkit would not be possible without the software it builds upon. We thank Manuvir Das for providing us his one-level flow analysis tool. We thank the developers of the AST toolkit at Microsoft Research, and Manuel Fähndrich for providing us his OCaml interface to the AST toolkit. Additionally, we have made good use of the publically available OCaml language, the Simplify and Vampyre theorem provers, and the BDD libraries of CMU and Colorado.

Thanks also to the members of the Software Productivity Tools research group at Microsoft Research for many enlightening discussions on program analysis, programming languages and device drivers. Finally, thanks to Jim Larus, who initially suggested device drivers as an interesting application domain.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL '02*. ACM, January 2002.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[3] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2001.

[4] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 268–283. Springer-Verlag, 2001.

[5] T. Ball, A. Podelski, and S. K. Rajamani. On the relative completeness of abstraction refinement. Technical Report MSR-TR-2001-106, Microsoft Research, 2001.

[6] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[7] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[8] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE 01: Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103. ACM, 2001.

[9] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[10] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[11] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[12] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[14] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.

[15] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *LICS 01: Symposium on Logic in Computer Science*, 2001.

[16] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report Research Report 159, Compaq Systems Research Center, December 1998.

[17] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *ICSE 01: Software Engineering*, pages 177–187, 2001.

[18] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.

[19] D. Evans. Static detection of dynamic memory errors. In *PLDI '96*, pages 44–53. ACM, May 1996.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*. ACM, January 2002.

[22] G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.

[23] R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[24] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[25] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL '02*. ACM, January 2002.

[26] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[27] V. Rusu and E. Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS 99: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, pages 178–192. Springer-Verlag, 1999.

[28] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 443–454. Springer-Verlag, 1999.

[29] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[30] M. Y. Vardi and P. Wolper. An automata theoretic apporach to automatic program verification. In *LICS 86: Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1996.