Software Verification with BLAST*

Thomas A. Henzinger,, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre[2]

[1] UC Berkeley
{tah,jhala,rupak}@eecs.berkeley.edu
[2] LaBRI, Université de Bordeaux, France
sutre@labri.u-bordeaux.fr

1 Introduction

BLAST (Berkeley Lazy Abstraction Software Verification Toolkit) is a software model-checker for C programs that is based on the property-driven construction and model-checking of software abstractions. At a high level, the BLAST algorithm follows the following loop for counterexample driven abstraction refinement [2, 3, 14]:

Step 1 ("abstraction") A finite set of predicates is chosen, and an abstract model of the given program is built automatically as a finite or push-down automaton whose states represent truth assignments for the chosen predicates.

Step 2 ("verification") The abstract model is checked automatically for the desired property. If the abstract model is error-free, then so is the original program (**return** "program correct"); otherwise, an abstract counterexample is produced automatically which demonstrates how the model violates the property.

Step 3 ("counterexample-driven refinement") It is checked automatically if the abstract counterexample corresponds to a concrete counterexample in the original program. If so, then a program error has been found (**return** "program incorrect"); otherwise, the chosen set of predicates does not contain enough information for proving program correctness and new predicates must be added. The selection of such predicates is automated, or at least guided, by the failure to concretize the abstract counterexample [3].

Goto Step 1.

BLAST short-circuits the loop from abstraction to verification to refinement, integrating the three steps tightly through "lazy abstraction" [11]. This integration can offer significant advantages in performance by avoiding the repetition of work from one iteration of the loop to the next [10].

Intuitively, lazy abstraction proceeds as follows. In Step 3, call the abstract state in which the abstract counterexample fails to have a concrete counterpart,

* This work was supported in part by the NSF ITR grant CCR-0085949, the NSF Theory grant CCR-9988172, the DARPA PCES grant F33615-00-C-1693, the MARCO GSRC grant 98-DT-660, the SRC contract 99-TJ-683.003 and a Microsoft Research Fellowship.

the *pivot state*. The pivot state suggests which predicates should be used to refine the abstract model. However, instead of building an entire new abstract model, we refine the current abstract model "from the pivot state on." Since the abstract model may contain loops, such *refinement on demand* may, of course, refine parts of the abstract model that have already been constructed, but it will do so only if necessary; that is, if the desired property can be verified without revisiting some parts of the abstract model, then our algorithm succeeds in doing so. The algorithm integrates all three steps by constructing and verifying and refining *on-the-fly* an abstract model of the program, until either the desired property is established or a concrete counterexample is found. Upon termination with the outcome "program correct," the proof is not an abstract model on a global set of predicates, but an abstract model whose predicates change from state to state.

We now describe the algorithm in more detail. Internally, programs are represented as *control flow automata (CFA)*, which are the control flow graphs, but with the operators on the edges. The lazy abstraction algorithm is composed of two phases. In the forward-search phase, we build a *reachability tree*, which represents a portion of the reachable, abstract state space of the program. Each node of the tree is labeled by a vertex of the CFA and a boolean formula, called the *reachable region*, constructed as a combination of a finite set of *abstraction predicates*. Edges of the tree correspond to edges of the CFA, and are labeled by the corresponding basic block or assume predicate. Each path in the tree corresponds to a path in the CFA. The reachable region of a node describes the reachable states of the program in terms of the abstraction predicates, assuming execution follows the sequence of instructions labeling the edges from the root of the tree to the node. If we find that an error node is reachable in the tree, then we go to the second phase, which checks if the error is real or results from our abstraction being too coarse (i.e., if we lost too much information by restricting ourselves to a particular set of abstraction predicates). In the latter case, we ask a theorem prover to suggest additional abstraction predicates which rule out that particular spurious counterexample. By iterating the two phases of forward search and backwards counterexample analysis, different portions of the reachability tree will use different sets of abstraction predicates. Moreover, from the reachability tree constructed by Blast, invariants that are sufficient to prove the safety property can be mined, and a short, formal, easily checkable proof (a la proof-carrying code [13]) can be constructed [10].

Blast has successfully verified and found violations of safety properties of large device driver programs. A beta version of Blast has been released and is available from http://www.eecs.berkeley.edu/~tah/blast.

2 The Blast Implementation

The input to Blast is a C program and a safety monitor written in C; these are compiled into a CFA with a special error state. The lazy-abstraction algorithm runs on the CFA and returns either a genuine error trace or a proof of correctness (or fails to terminate). The proof is encoded in ELF notation as in

proof-carrying code [13], and can be checked by an existing PCC implementation. Our handling of C features follows that of [1]. We handle all syntactic constructs of C, including pointers, structures, and procedures (leaving the constructs not in the predicate language uninterpreted). However, we model integer arithmetic as infinite-precision arithmetic (no wraparound), and we assume a *logical* model of the memory. Currently we handle procedure calls using an explicit stack and do not handle recursive functions.

Our tool is written in Objective Caml, and consists of two main parts: (1) an implementation of the lazy abstraction algorithm, which works on a generic symbolic abstraction structure and a generic refinement function, and (2) the symbolic abstraction structure and refinement operator for C. The latter is made up of two parts: (a) the C front end, for which we use the CIL compiler infrastructure [12], which converts a program to its CFA, and (b) a module that contains the data structures for C regions as well as the concrete predecessor, abstract successor, and counterexample refinement functions. The boolean formulas over predicates that represent data regions are stored as BDDs [15] to get a canonical sum-of-product form for formulas. The BDD representation also allows easy boolean manipulation and inclusion checking. Finally, we use the theorem prover Simplify [6] for abstract-successor computations and inclusion checks, and the (slower) proof-generating theorem prover Vampyre [16] for abstraction refinement, where proofs are required. The proofs are generated in LF format and uses a standard PCC implementation for encoding into binary.

In order to be practical, the tool uses several optimizations. The cost is dominated by the cost of theorem proving, so we extensively optimize calls to the theorem prover. Instead of the most precise abstraction for the successor operation [5], we compute a fast Cartesian abstraction (that is usually strong enough to prove the properties). In the computation of $\widehat{post}$, we check if a predicate p is affected by a statement s, and invoke theorem prover calls only on the subset of predicates for which $wp(p, s) \neq p$. We remove predicates relating values not in the current scope, to do this without losing information, we use the theorem prover data structures to add additional useful predicates. We apply a set of program analysis optimizations up front: these include interprocedural constant propagation, dead code elimination, and redundant variable elimination. We also implement a simple program slicing based on a cone of influence on variables appearing in conditionals.

BLAST is essentially a *whole program analysis*, it works best when all the code is available. Frequently, analysis requires only *models* of certain functions, and not their actual implementation. For example, in checking for locking behavior in the Linux kernel, one simply requires a model of the `spin_unlock` function that sets a particular state variable of the specification, and not the actual assembly code that implements locking in the kernel. We model such kernel calls using stub functions. Otherwise, for a function whose return value is dropped, if the body of a called function is not available, BLAST makes the optimistic assumption that the function is of no relevance to the particular property being checked, and so

no predicate values are updated (but a warning message with the names of the unknown functions is printed).

Acknowledgments. We thank George Necula and Westley Weimer for various discussions and for providing support with Cil.

References

1. 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.
2. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
4. J. Corbett, M. Dwyer, John Hatcliff, Corina Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, pages 439–448, 2000.
5. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 99: Computer-Aided Verification*, LNCS 1633, pages 160–171. Springer-Verlag, 1999.
6. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover. http://research.compaq.com/SRC/esc/Simplify.html.
7. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
8. 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.
9. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.
10. T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 526–538. Springer-Verlag, 2002.
11. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
12. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, LNCS 2304, pages 213–228. Springer-Verlag, 2002.
13. G.C. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.
14. H. Saidi. Model checking guided abstraction and analysis. In *SAS 00: Static-Analysis Symposium*, pages 377–396. LNCS 1824, Springer-Verlag, 2000.
15. F. Somenzi. Colorado University decision diagram package. 1998.
16. Vampyre. http://www.eecs.berkeley.edu/~rupak/Vampyre.