Software Verification with BLAST*

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

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

Introduction. BLAST (the Berkeley Lazy Abstraction Software verification Tool) is a verification system for checking safety properties of C programs using automatic property-driven construction and model checking of software abstractions. BLAST implements an abstract-model check-refine loop to check for reachability of a specified label in the program. The abstract model is built on the fly using predicate abstraction. This model is then checked for reachability. If there is no (abstract) path to the specified error label, BLAST reports that the system is safe and produces a succinct proof. Otherwise, it checks if the path is feasible using symbolic execution of the program. If the path is feasible, BLAST outputs the path as an error trace, otherwise, it uses the infeasibility of the path to refine the abstract model. BLAST short-circuits the loop from abstraction to verification to refinement, integrating the three steps tightly through "lazy abstraction" [5]. This integration can offer significant advantages in performance by avoiding the repetition of work from one iteration of the loop to the next.

We now describe the algorithm in more detail. Internally, C programs are represented as control flow automata (CFA), which are control flow graphs with operators on 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 formula, called the reachable region, constructed as a boolean combination of a finite set of abstraction predicates. Initially the set of abstraction predicates is empty. The edges of the tree correspond to edges of the CFA and are labeled by basic program blocks or assume predicates. 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 new abstraction predicates which rule out that particular spurious counterexample. The program is then refined locally by adding the new abstraction predicates only in the smallest subtree containing the spurious error; the

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

search continues from the point that is refined, without touching the part of the reachability tree outside that subtree.

Thus the benefits are three-fold. First, we only abstract the reachable part of the state space, which is typically much smaller than the entire abstract state space. Second, we are able to have different precisions at different parts of the state space, which effectively means having to process fewer predicates at every point. Third, we avoid redoing the model checking over parts of the state space that we know are free of error from some coarser abstraction. Moreover, from the reachable set constructed by BLAST, invariants that are sufficient to prove the safety property can be mined, and a short, formal, easily checkable proof of correctness can be constructed [4]. BLAST has successfully verified and found violations of safety properties of large device driver programs up to 60,000 lines of code. A beta version of BLAST has been released and is available from `http://www.eecs.berkeley.edu/~tah/blast`.

Implementation. The input to BLAST is a C program and a safety monitor written in C. The program and the monitor are compiled into a single program with a special error location that is reachable iff the program does not satisfy the safety property. The lazy-abstraction algorithm runs on this program and returns either a genuine error trace or a proof of correctness (or fails to terminate). The proof is encoded in binary ELF format as in proof-carrying code [6]. Our tool is written in Objective Caml, and uses the CIL compiler infrastructure [7] as a front end to parse C programs. 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 wrap-around), and we assume a logical model of the memory. In particular, we disallow casting that changes the "layout pattern" of the memory, disallow partially overlapped objects, and assume that pointer arithmetic in arrays respects the array bound. Currently we handle procedure calls using an explicit stack and do not handle recursive functions.

Our implementation works on a generic *symbolic abstraction structure* which is an internal representation that provides a symbolic interface suitable for model checking, namely a representation of sets of states ("regions"), and functions to compute the concrete and abstract predecessor and successor regions, analyze counterexamples, and refine abstractions.

A C program is represented internally as a CFA. A region is a tuple of CFA state (location), data state, and stack state. We represent the CFA state explicitly, but represent the data state symbolically as boolean formulas over the abstraction predicates. The stack state is a sequence of CFA states. The boolean formulas are stored in canonical form as BDDs.

Given a region and an edge of the CFA, the concrete successor and predecessor operators are implemented using syntactic strongest postcondition and weakest precondition operators, respectively. Given a region, a set of abstraction predicates, and an edge of the CFA (an operation in the program), the symbolic abstract predecessor and successor operators compute an overapproximation of the concrete predecessor and successor sets representable using the abstraction predicates, by making queries to the decision procedures Simplify [3] or CVC [8].

Counterexample analysis is implemented by iterating the concrete predecessor or successor operators and checking for unsatisfiability. Finally, the refinement operator takes an infeasible counterexample trace (whose weakest precondition w.r.t. *true* is, by definition, unsatisfiable), and generates new abstraction predicates by querying a proof generating theorem prover (like Vampyre or CVC in proof generation mode) for a proof of unsatisfiability, and taking the atomic formulas appearing in the proof.

The lazy abstraction algorithm is implemented on top of the interface provided by the symbolic abstraction structure. It does not depend on internal data structures of the symbolic abstraction structure. The advantage of separating the model checking algorithms from the particular internal representation of the system is that we can reuse much of the code to build a model checker for different front ends (for example, for Java programs), or for different region representations. A clean symbolic abstraction structure interface also allows us to experiment with different model checking algorithms and heuristics.

Optimizations. 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. First, we compute a fast (linear in the number of predicates) abstract successor operation which is less precise than [2] but usually strong enough to prove the desired properties [5]. Moreover, when computing the abstract successor operator w.r.t. a statement s, we check if the predicate p is affected by the statement s (by checking if $p \neq wp(p, s)$, where wp is the weakest precondition operator), and invoke theorem prover calls only on the subset of predicates which are affected. Second, while constructing the weakest precondition, we only keep satisfiable disjuncts (disjuncts appear in the weakest precondition because of aliasing). Third, we remove predicates that relate variables not in the current program scope. To do this without losing information, we use the theorem prover data structures to add additional useful predicates. Apart from reducing the theorem proving burden, this optimization enables us to reach the fixpoint quicker. Fourth, the check for region inclusion is performed (without sacrificing precision) entirely at the boolean level by keeping the predicates uninterpreted.

We apply a set of program analysis optimizations up front: these include interprocedural conditional constant propagation, dead code elimination, and redundant variable elimination. We have also implemented simple program slicing based on the cone of influence on variables appearing in conditionals.

All the heuristics can be independently turned on or off through command line options; this allows us to experiment with several combinations. With these optimizations, BLAST routinely runs on several thousand lines of C code in a few minutes.

Experiences. Frequently, reachability 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` and `spin_lock` functions 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 manually using stub functions that implement the required behavior. We found nondeterministic choice to be a useful modeling tool (for example, to model that a function can return either 0 or 1 nondeterministically), so we added explicit support for nondeterministic choice. By default, if the body of a called function whose return value is dropped 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 is printed).

Sometimes the predicate discovery is not strong enough to find all predicates of interest and we take (as optional input) a file with programmer specified predicates; the syntax of predicates follows the C syntax for expressions. We find in our experiments that efficient discovery of good predicates is still an important issue. We have implemented various heuristics to find more predicates, including a scan of a counterexample trace to find multiple causes of unsatisfiability.

BLAST has been used to verify several large C programs [4]. Most of these programs are device driver examples from the Microsoft Windows DDK or from the Linux distribution. The properties we checked ranged from simple locking mechanisms to checking that a driver conforms to Windows NT rules for handling I/O requests. We have found bugs in several drivers, we have proved that other drivers correctly implement the specification.

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. 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.
3. D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover.
4. 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*, LNCS 2404, pages 526–538. Springer-Verlag, 2002.
5. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
6. G.C. Necula. Proof-carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.
7. 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.
8. A. Stump, C. Barrett, and D.L. Dill. CVC: A cooperating validity checker. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 500–504. Springer-Verlag, 2002.