Predicate Analysis with BLAST 2.7
(Competition Contribution)

Pavel Shved, Mikhail Mandrykin, and Vadim Mutilin

Institute for System Programming of the Russian Academy of Sciences
{shved,mandrykin,mutilin}@ispras.ru

Abstract. We present the software verification tool BLAST 2.7, which we submitted for the Competition on Software Verification. The tool is an improvement over BLAST 2.5, and its development is mostly targeted at its performance and usability in the Linux Driver Verification project. The paper overviews the tool and outlines our contribution to it.

1 Verification Approach

BLAST uses the CounterExample-Guided Abstraction Refinement approach (CEGAR) with "lazy abstraction" [1], a decision procedure to explore all possible paths from the entry point, abstracting away from the realizable memory states as far as possible to prove the unreachability of the error label. BLAST marks each location with a conjunction of predicates over program variables in a path-sensitive way, such conjunctions being overapproximations of the set of feasible memory states at the locations. Path validity is checked with *formula satisfiability solvers*; *interpolating provers* automatically retrieve predicates to track. The concepts of BLAST are more thorougly described in [1] and [2]. BLAST may also combine the predicate domain with lattice-based explicit-value dataflow analysis [3], which we used in the competition setup.

This has been implemented in BLAST 2.5, which was maintained by Dirk Beyer et al. [2]. In this paper, we present the improvements that we added to BLAST since the release of version 2.5 in 2008; we assigned version 2.7 to the competition release. Most of the improvements are merely more efficient implementations of already known algorithms.

The tool as of version 2.7 is capable to track states that may be expressed in terms of logical formulæ over atomic predicates that only contain linear (in)equalities over program variables, including aliases of pointers to scalar or structure variables. The analysis may be unsound if the unreachability proof requires reasoning about bit-vectors, bounded integers, or arrays.

2 Tool Architecture

The tool first converts the program into a set of per-function control flow automata with aid of the **CIL C frontend** (integrated into the tool). It converts

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 525–527, 2012.

the structure of C source code directly into OCaml memory structures, and helps to perform transformations that simplify the semantics of individual operators.

Having built the CFAs, BLAST starts the abstract-check-refine loop, inlining each function call it encounters on demand, and skipping recursive calls. In the forward search phase, it uses an SMTlib solver to compute the abstract post-condition for predicates (**CVC3** is shipped alongside the tool, but any other decent SMTlib solver would work), and updates *symbolic execution lattice* elements [3]. The predicates are stored as BDDs over atomic predicate symbols. A potential error path is converted to a path formula (weakest precondition of each operator starting from the end with explicit substitution, see section 5.3 of [1]). It is checked for satisfiability with SMTlib solver, then filtered with multiple SMTlib solver calls to get unsatifiability cores, which then undergo Craig interpolation with **CSIsat** (or any other tool that supports the FOCI format) to get the predicates to track.

Both forward exploration and path analysis are supplemented with inter-procedural points-to may-alias information provided by an subset-based Andersen analysis with BDDs as a storage and querying mechanism.

BLAST is implemented in OCaml.

3 Tool Improvements and Benchmarking

As BLAST 2.5 demonstrated some success in verifying drivers, we used it in the Linux Driver Verification (LDV) project [4], and improved it further to make the tool faster for Linux drivers. The rules instrumented into drivers used states expressed as simple integers, but the drivers themselves used structures and pointers to maintain data flow; therefore, we focused on improving implementations of the existing theoretical achievements, and did not try to extend the abstract domain.

In version 2.6, we improved folmulæ conversion between the internal OCaml representation and SMTlib format, making its overhead negligible, tuned the CVC3 solver to work faster for quantified formulæ, decreased the asymptotics of pre-interpolation trace filtering from $O(N)$ to $O(\log N)$ solver calls, and implemented $stop^{sep}$ and *merge-pred-join* for combining predicate analysis with lattices. We thorougly described these and many other improvements of 2.6 over 2.5 in [5]. The speedup we achieved on Linux drivers, compared with version 2.5, ranged from a factor of 5–8 on average to 30 on the most complex drivers. Our tool performed well on all driver-related benchmarks.

In the competition version 2.7, we also dramatically improved alias and structure analysis that are used to generate additional variable updates at assignment preconditions. We noticed that updates of indexed variables that are not used in the bottom part of the path formula were useless, and, while they would be ruled out by solvers anyway, BLAST spent a lot of time generating them. We revamped the generation algorithms so that only the variables that are in the already built part of a formula are considered as potential aliases or targets for structure updates; this made alias analysis overhead negligible. The new analysis is sound,

but sometimes incomplete for variable-depth shapes. It improved the results on `list-properties` benchmarks, but the `heap-manipulation` benchmarks are analyzed with errors due to abuse of low-level-style accesses to structure fields via casts and raw pointer shifts.

Other benchmarks, such as `SystemC` or synthetic `locks`, involved state explosions that should be mitigated by verification algorithms that automatically merge states without loss of precision. BLAST does not merge paths with different predicate states assigned, so it times out on most of such benchmarks, which more recent tools should pass.

4 Downloading and Using BLAST

To use the tool, download binaries, unpack, add the `bin/` folder to your `PATH`, and run: `ocamltune pblast.opt -alias bdd -enable-recursion -noprofile -cref -sv-comp -lattice -include-lattice symb FILE_NAME.c`. Download: http://forge.ispras.ru/attachments/download/1157/blast-2.7-bin-x86_64.tgz.

Visit http://forge.ispras.ru/projects/blast/ to get the source code and 32-bit binaries. BLAST is licensed under Apache-2.0, and *all* external tools it relies on during compilation or at runtime are free software.

The verdict and the error trace, if any, are written to standard output. For more information on the tool usage, please, refer to the `README` file.

The binary distribution of the tool does not require external tools except for the Perl interpreter, C and C++ runtime libraries. BLAST is compatible with most modern Linux distributions, including Ubuntu 8.04 or newer.

Acknowledgements. BLAST 2.7 was prepared as a part of the Linux Driver Verification project with the help of our colleagues at ISPRAS. A number of people contributed to BLAST, including its former maintainers Dirk Beyer, Rupak Majumdar, Ranjit Jhala, and Thomas Henzinger, and the others mentioned in the `README` file.

References

1. Henzinger, T.A., Jhala, R., Majumdar, R.: Lazy abstraction. In: Symposium on Principles of Programming Languages, pp. 58–70. ACM Press (2002)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST: Applications to software engineering. Int. J. Softw. Tools Technol. Transf. 9(5), 505–525 (2007)
3. Fischer, J., Jhala, R., Majumdar, R.: Joining dataflow with predicates. SIGSOFT Softw. Eng. Notes 30, 227–236 (2005)
4. Khoroshilov, A., Mutilin, V., Novikov, E., Shved, P., Strakh, A.: Towards an open framework for C verification tools benchmarking. In: Proceedings of PSI (2011)
5. Shved, P., Mutilin, V., Mandrykin, M.: Static verfication "under the hood": Implementation details and improvements of BLAST. In: Proceedings of SYRCoSE, vol. 1, pp. 54–60 (2011)