Vienna Verification Tool: IC3 for Parallel Software
(Competition Contribution)

Henning Günther, Alfons Laarman(✉), and Georg Weissenbacher

TU Wien, Vienna, Austria
alfons@laarman.com

Abstract. Recently proposed extensions of the IC3 model checking algorithm offer a powerful new way to symbolically verify software. The Vienna Verification Tool (VVT) implements these techniques with the aim to tackle the problem of parallel software verification. Its SMT-based abstraction mechanisms allow VVT to deal with infinite state systems. In addition, VVT utilizes a coarse-grained large-block encoding and a variant of Lipton's reduction to reduce the number of interleavings. This paper introduces VVT, its underlying architecture and use.

1 Verification Approach

VVT is an implementation of the CTIGAR approach [2], an SMT-based IC3 algorithm [3] incorporating Counterexample-Guided Abstraction Refinement (CEGAR) [5], thus enabling the verification of infinite-state systems. The underlying abstraction-refinement scheme follows the IC3 paradigm; it does not require an unwinding of the transition relation. To handle parallel programs, VVT uses a large-block encoding [8] that preserves all relevant partial interleavings by applying a novel dynamic variant of Lipton's reduction [12].

2 Software Architecture

VVT uses a modular approach to verification: a collection of separate tools instrument and translate the input, communicating via standard data formats such as LLVM bitcode [4] and the SMTlib format [1]. Figure 1 provides an overview.

The verification process begins by compiling the C file into LLVM bitcode using CLang. The LLVM IR has a precise semantics and comprises only a small number of instructions, thus reducing the complexity of the verifier. The increase in size resulting from the translation into bitcode is mitigated by subsequent

This work is supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 954–957, 2016.
DOI: 10.1007/978-3-662-49674-9_69

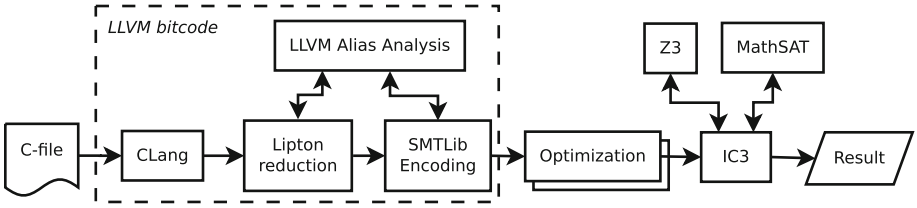

Fig. 1. Architecture

reduction steps. A separate tool implementing a variant of Lipton's reduction [12] identifies large blocks that can be executed atomically. These blocks are delimited by instrumenting the code with 'yield' function calls, indicating the relevant context switches. Our novel dynamic reduction method avoids static analysis[1] by expressing reduction conditions as branches. At each intermediate step the LLVM tool chain is used to optimize the bitcode (not shown in the figure).

Next, the `vvt-enc` tool translates the instrumented bitcode into an SMTlib-based format, encoding the transition relation of the program. It uses (linear) integer arithmetic to encode bit vectors to facilitate interpolation and employs alias-analysis techniques in order to keep the transition relation as small as possible. To finalize the encoding, the `vvt-opt` tool deploys a number of optimization techniques including program slicing (removing irrelevant parts of the transition relation), expression simplification and a value-set analysis (to identify constant expressions).

The last step is the actual verification with the `vvt-verify` tool. It uses Z3 [6] for IC3 consecution calls [3] and MathSAT [7] for interpolation-based refinement. To rapidly find counterexamples, VVT runs a small portfolio with the BMC tool `vvt-bmc` [11] on the same encoding (not shown in the figure), taking advantage of the modularity of the tool chain.

3 Strengths and Weaknesses

VVT primarily targets the verification of infinite parallel programs. Unlike BMC tools, the approach is complete and does not depend on a complete unrolling of the transition relation thanks to the underlying IC3 algorithm. The SMT-based abstraction-refinement scheme further extends the capabilities of the tool to infinite-state systems. Finally, parallelism is supported by the reductions applied to the transition relation.

Our experiments show that VVT yields good results on almost all instances of the concurrency category of the Software Verification Competition (SVCOMP) 2016. The verification results for integer/control-flow programs demonstrate that the abstraction-refinement mechanisms work well in practice.

[1] A lack of good static analysis is a bottleneck for obtaining powerful reductions in software model checking [9].

VVT currently does not implement rely-guarantee reasoning, and is therefore unable to handle an infinite number of threads. Furthermore, the lack of an interpolating decision procedure for arrays limits the applicability of the tool for programs with arrays to those cases where the size of the arrays can be determined statically.

VVT generates concrete counterexample traces, but does not yet map the LLVM instructions to locations in the original source code.

4 Tool Setup and Configuration

The Vienna Verification Tool is open source and distributed under the GPL license. Both the source code and the packaged version 0.1 submitted to SVCOMP 2016 can be found at the VVT website [10].

Installation. VVT v0.1 [10] requires packages LLVM 3.5 and CLang 3.5, which are available via standard package managers (APT, RPM, etc.) on many systems.

The command `vvt-svcomp-bench.sh <FILE>` starts the entire verifier tool chain (see Fig. 1), where `<FILE>` is the C or C++ file to be verified.

Participation Statement. For the SVCOMP 2016, we enlist VVT for participation in the categories *Integers and Control Flow* and *Concurrency*. In the former, we opt out of the sub-categories: *recursive*, *loops*, *product lines*, and *sequentialized*. We also opt out VVT of the other (unmentioned) categories.

5 Software Project and Contributors

VVT is developed by the Formal Methods in Systems Engineering (FORSYTE) group of the Vienna University of Technology. For more information, contact Henning Günther. Bug reports and feature requests can be submitted via the VVT website [10].

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds,) SMT Workshopp (2010)
2. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 829–846. Springer, Heidelberg (2014)
3. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
4. Lattner, C., Adve, V.: The LLVM Instruction Set and Compilation Strategy. Technical report UIUCDCS-R-2002-2292, University of Illinois (August 2002)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. LNCS, pp. 154–169. Springer, Heidelberg (2000)

6. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
8. Beyer, D., et al. Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
9. Flanagan, C., Qadeer, S.: Transactions for software model checking. Electron. Notes Theor. Comput. Sci. **89**(3), 518–539 (2003)
10. Günther, H.: VVT website. http://vvt.forsyte.at, last visited: January 2016
11. Günther, H., Weissenbacher, G.: Incremental bounded software model checking. In: SPIN, pp. 40–47. ACM (2014)
12. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975)