Vienna Verification Tool:
IC3 for Parallel Software

Henning Günther⋆, Alfons Laarman, and Georg Weissenbacher

TU Wien, Austria⋆⋆⋆

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 (CEGAR) [5], thus enabling the verification of infinite-state systems. The underlying abstraction-refinement scheme follows the IC3 paradigm, as 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 [10].

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]. Fig. 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 reduction steps. A separate tool implementing a variant of Lipton's reduction (see footnote 1) uses static analysis to identify large blocks that can be executed atomically. These blocks are delimited by instrumenting the code with 'yield'
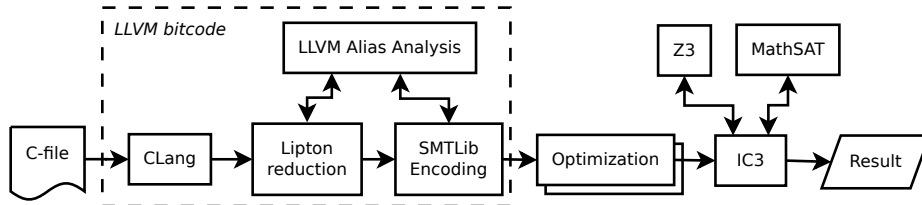

  ⋆ Corresponding author

Fig. 1. Architecture

function calls, indicating the relevant context switches. A dynamic reduction is achieved by means of branching statements. 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. 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` [9] on the same encoding, 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.

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. The source code can be found on GitHub.[1]
Installation. The packaged binaries of VVT v0.1 are available at

https://www.dropbox.com/s/9ms4e9ye0hnems1/vvt.tar.xz .

The required packages LLVM 3.5 and CLang 3.4 need to be installed on the system and are available via a standard package manager (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. Bug reports can be submitted at the VVT GitHub page.[1]

References

1. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *SMT Workshop*, 2010.
2. Johannes Birgmeier, Aaron Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *CAV*, volume 8559 of *LNCS*, pages 829–846. Springer, 2014.
3. Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.
4. Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech Report UIUCDCS-R-2002-2292, Univ. of Illinois, Aug 2002.
5. Edmund et al. Clarke. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.
6. Leonardo de Moura and Nikolaj Børner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. Alessandro Cimatti et al. The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *TACAS*, volume 7795 of *LNCS*. Springer, 2013.
8. Dirk Beyer et al. Software model checking via large-block encoding. In *FMCAD*, pages 25–32. IEEE, 2009.
9. Henning Günther and Georg Weissenbacher. Incremental bounded software model checking. In *SPIN*, pages 40–47. ACM, 2014.
10. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

---

[1] VVT can be downloaded at https://github.com/hguenther/vvt. The tool to infer atomic blocks is available at https://github.com/alaarman/Liptonizer