SVISS: Symbolic Verification of Symmetric Systems*

Thomas Wahl[1], Nicolas Blanc[1], and E. Allen Emerson[2]

[1] Computer Systems Institute, ETH Zurich, Switzerland
[2] Department of Computer Sciences, The University of Texas at Austin, USA

Abstract. SVISS is a flexible platform for incorporating efficient symmetry reduction into symbolic model checking. The tool comes with an extensive `C++` library for system modeling using BDDs and a rich CTL-based model checking engine. Applications range from communication protocols to computer hardware and multi-threaded software. We believe SVISS to be the first symbolic tool to exploit symmetry in concurrent device-driver verification, which is vital in operating system design.

1 Introduction

Symmetry reduction has proved to effectively curb the complexity of model checking finite-state multi-process systems. Provided the transition relation of the system is invariant under permutations of the participating processes, states that are identical up to permutations can be collapsed into an equivalence class, known as an *orbit*. This can reduce the number of states that need to be kept in memory from exponential to polynomial in the number of processes.

In contrast to its immediate success with explicit-state model checkers such as MURφ [7], symmetry reduction of a system given symbolically as a Binary Decision Diagram (BDD) was first thought to be infeasible in practice due to the *orbit problem*: the BDD representing the symmetry equivalence relation is of intractable size [3]. In this paper, we present a symbolic model checker with symmetry reduction that never builds this BDD and thus avoids the orbit problem: SVISS (historically, "*S*ymbolic *V*erification of *I*nvariants of *S*ymmetric *S*ystems") implements, to our knowledge, the first *efficient* symbolic realization of symmetry reduction, by dynamically mapping each encountered state to a unique representative of its orbit [5].

SVISS comes with a rich `C++` library for constructing transition relations. The benefit of a library in a widely known programming language over a specialized input language is flexibility: the library has been used to model systems as diverse as asynchronous communication protocols [5], Boolean abstractions of concurrent software (see section 3), synchronous parallel programs [4], and finite-state machine descriptions of computer hardware. The penalty for this flexibility is that by using `C++` constructs outside the library, the user can circumvent

* Work supported by the Swiss National Science Foundation under grant 200021-109594, by ETH research grant TH-21/05-1, and by NSF grant CCR-020-5483.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 459–462, 2008.

restrictions that ensure symmetry, which are therefore not enforced by the tool (a weakness that SVISS shares with other tools exploiting symmetry including MURφ [7]).

SVISS especially supports experimentation. An input file contains no property specifications. Instead, once the transition relation is built, the tool repeatedly requests CTL-like formulas at a prompt and performs global model checking by computing the set of states satisfying the formula. If the set is small, it can be visualized in a compact format. The intended utility of this feature is to increase confidence in the model by inspecting the set of initial, bad or reachable states of a small instance of a parameterized system.

2 Tool Description and Usage

The state space of the design under investigation is described in SVISS through model parameters, constants, and program variables. The variables can be of type Boolean, finite range, enumeration, record and array. The `C++` library offers routines to access these variables, further Boolean operators, simple linear arithmetic, and a set of specialized functions for transition relation construction. An example illustrating the use of the library is provided on the tool website: `http://www.inf.ethz.ch/~wahlt/Sviss`.

Variables are either global or belong to a *symmetry block*. Each block comprises the local variables of processes forming a symmetric factor of the state space (such as a block of readers and a block of writers in the Readers-Writers problem). A block also specifies the number of replicated components and the symmetry group that is to act within it. SVISS offers reduction with respect to full (arbitrary permutations of components) and rotational symmetry (cyclic shifts), specified by the user individually for each block. Global variables that store process indices (such as a token variable in a resource allocation protocol) are allowed and treated specially by the reduction algorithms.

SVISS first compiles a system model to a customized executable. The model may leave some parameters unspecified, such as the number of process components. These parameter, as well as CTL specifications, can conveniently be supplied later at the command line of the executable or at a prompt. This greatly facilitates experimentation with different parameters and specifications.

SVISS computes the set of states corresponding to an input formula, which can be written in a dialect of CTL, augmented by past-time temporal operators, with or without frontier set optimization. The computation can be done (i) ignoring symmetry, (ii) using dynamic symmetry [5], (iii) using the *orbit relation* [3] and (iv) by way of *multiple representatives* [3].[1] If the result is neither empty nor equal to the entire state space, the set of states (or a few elements of it) can be enumerated. For experimental purposes, SVISS further supports the computation of a set's cardinality, of the corresponding set of symmetry-representative states, and of the corresponding set of symmetry-equivalent states (i.e. the set's orbit).

[1] On average, efficiency seems to diminish in the order (ii), (i), (iv), (iii).

SVISS possesses a specialized operator INV, which checks invariant conditions after each step during symbolic reachability analysis, either forward (from *init*) or backward (from *error*). Upon failure, the tool prints an error trace in terms of the original program variables.

SVISS uses the CUDD decision diagram package [9] for BDD manipulations.

3 Applications of SVISS

SVISS has been applied successfully to communication and locking protocols and to systems parameterized by the number of processes, in one fell swoop over a finite range of the parameter. Quantitative results from these experiments are available in the cited literature [5,4], which also compares the performance of SVISS's algorithms with alternative approaches to exploiting symmetry.

In this paper we share our experiences of applying SVISS to concurrent Linux device-driver software. A driver is confronted with a set of processes representing users, the operating system environment and external events. We used DDVerify [11] to obtain a (coarse) Boolean abstraction of the driver software with about 10-12 Boolean predicates per driver. All abstract models contain errors (often spurious), at depths ranging from 100 to 200 instructions.

The histograms in figure 1 show time (top left, log-scale) and space (bottom left) demands for safety-checking eight abstract models with a fixed number of

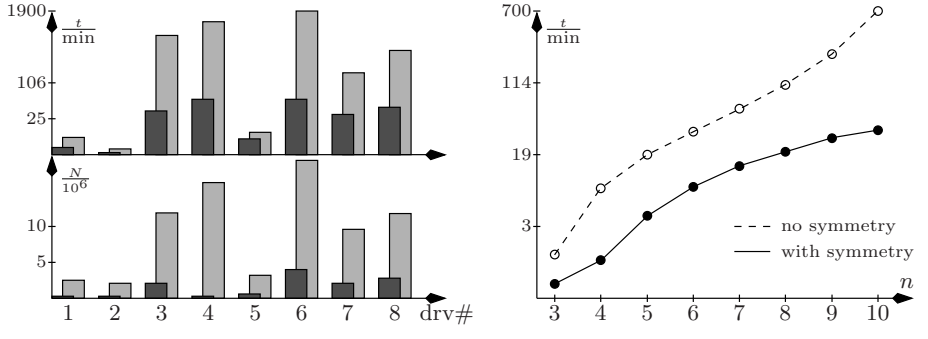

Fig. 1. Comparison of resource demands without and with symmetry across drivers for ten processes (left) and across numbers of components for driver # 3 (right)

ten processes, ignoring (light gray) and exploiting (dark gray) symmetry dynamically [5]. We see tremendous memory savings thanks to symmetry, in all cases.[2] The same holds for the run-time, with a few exceptions (e.g. drivers # 1, 2, 5); the exceptions correspond to shallow errors. The deeper the exploration, the greater the time and space savings of symmetry reduction. A similarly widening gap can be observed for a growing number n of components (graph on the

[2] "Memory" = peak number N of allocated BDD nodes. Experiments conducted on a 3 GHz Intel™ Pentium™ 4 dual-core processor, 2 GB of main memory.

right, only time is shown). After reaching a certain number of allocated BDD nodes, the cost of computing a transition image far exceeds that of symmetry-canonizing the set of successor states. The image computations benefit from a small set of representative states in the case of symmetry.

4 Related Work and Conclusions

Distinguished examples of *explicit-state* model checkers using symmetry include Murφ [7], SMC [8] and Zing [1]. Due to the enumeration, these tools are limited to systems with a manageable number of reachable states. Present-day (partially) BDD-based model checkers that offer symmetry reduction include UPPAAL [6], RULEBASE [2] and RED [10]. To escape the orbit problem, these tools usually fall back on approximate reduction strategies.

Concurrent software verification is still in its infancy. Symmetry reduction can help this effort by (i) increasing the depth up to which programs can be explored in reasonable time, (ii) increasing the number of abstraction-refinement iterations, each of which entails more predicates and thus more resource needs than its predecessor, and (iii) increasing the number of processes to which, say, a device driver can be exposed for verification. A future step is to integrate Sviss fully into an abstraction-refinement framework based on Boolean programs.

References

1. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 484–487. Springer, Heidelberg (2004)
2. Barner, S., Grumberg, O.: Combining symmetry reduction and Under Approximation for symbolic model checking. In: FMSD 2005 (2005)
3. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. In: FMSD 1996 (1996)
4. Emerson, E.A., Trefler, R.J., Wahl, T.: Reducing Model Checking of the Few to the One. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 94–113. Springer, Heidelberg (2006)
5. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005)
6. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to Uppaal. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, Springer, Heidelberg (2004)
7. Melton, R., Dill, D.L.: Murφ Annotated Reference Manual, rel. 3.1., `http://verify.stanford.edu/dill/murphi.html`
8. Sistla, A.P., Gyuris, V., Emerson, E.A.: Smc: a symmetry-based model checker for verification of safety and liveness properties. In: ACM ToSEM 2000 (2000)
9. Somenzi, F.: The CU Decision Diagram Package, release 2.3.1, University of Colorado at Boulder, `http://vlsi.colorado.edu/~fabio/CUDD/.`
10. Wang, F., Schmidt, K., Yu, F., Huang, G.-D., Wang, B.-Y.: Bdd-based safety-analysis of concurrent software with pointer data structures using graph automorphism symmetry reduction. In: IEEE ToSE 2004 (2004)
11. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE 2007 (2007)