Demiurge: A SAT-Based Synthesis Tool

Robert Könighofer[1] and Martina Seidl[2]

[1]Institute for Applied Information Processing and Communications, Graz University of Technology, Austria
[2]Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria.

***Abstract*—This document describes Demiurge, an open-source tool for synthesizing reactive systems from safety specifications using decision procedures for the satisfiability of quantified and unquantified Boolean formulas, as submitted to SyntComp 2014. Demiurge is not just a synthesis tool, it is also an extendable framework: New synthesis algorithms and optimizations can easily be integrated in new back-ends, reusing existing infrastructure like the parser, interfaces to SAT- and QBF solvers, existing procedures to extract circuits from strategies, etc. We describe the basic architecture and briefly sketch the working principle of the existing back-ends. Details to the implemented algorithms, optimizations, and first experiments can be found in [1].**

I. Introduction

Demiurge follows the traditional game-based approach to synthesis of reactive systems from safety specifications. The specification is seen as a game between two players: the environment player controls the inputs, and the system player controls the outputs of the system. The goal of the system player is to satisfy the specification, i.e., visit only safe states, independent of the environment behavior. In a first step, a so-called *winning region* is computed. The winning region is the set of all states from which the system player can enforce to satisfy the specification. In a second step, a *winning strategy* is derived from the winning region. For every (current) state and input, the winning strategy defines a set of outputs that are okay for satisfying the specification. The last step is to implement this strategy in a circuit, where a concrete choice for the outputs has to be made for every state and input.

In order to achieve acceptable scalability, it is important to implement synthesis algorithms symbolically, i.e., by manipulating formulas instead of enumerating states. In synthesis, these symbolic algorithms are, in turn, usually implemented with Binary Decision Diagrams (BDDs). One reason is that solving games inherently involves dealing with quantifier alternations, and BDDs offer both kinds of quantification. However, BDDs also have their scalability issues. On the other hand, there have been enormous performance improvements in decision procedures for the satisfiability of formulas over the last years and decades. This has lead to efficient tools like SAT- and QBF (Quantified Boolean Formulas) solvers. Demiurge leverages this development by implementing symbolic synthesis algorithms using such SAT- and QBF solvers.

Demiurge implements different synthesis algorithms in different back-ends. The back-ends can be run in different modes (optimizations and heuristics enabled and disabled) and with different solvers. The *learning-based back-end* computes the winning region with computational learning using either a QBF- or a SAT solver. It also implements heuristics to exploit reachability information. The *parallel back-end* implements the same learning-based method with several threads refining the winning region in parallel. The *template-based back-end* uses a QBF solver to compute the winning region as instantiation of a template for a CNF formula over the state variables. Details to these back-ends can be found in [1]. All these back-ends can be used with several methods to extract circuits from the computed winning region. This includes methods based on QBF certification [7] as well as computational learning [3] using SAT- or QBF solvers. ABC is used in a post-processing step to reduce the circuit size. Further back-ends include a re-implementation of [6], and an approach based on a reduction to Effectively Propositional Logic (EPR) [1].

The modular architecture makes Demiurge easily extendable with new algorithms and optimizations. A lot of infrastructure like interfaces to solvers and entire steps of the synthesis procedure (like extracting a circuit from a winning region) can be reused. At the moment, Demiurge contains uniform interfaces to the APIs of Minisat, Lingeling, PicoSat, and DepQBF (with and without the QBF preprocessor Bloqqer [8]). The interface to DepQBF also supports incremental QBF solving [5]. Interfaces to SAT- and QBF-solvers supporting DIMACS or QDIMACS format are available as well. Furthermore, Demiurge interfaces ABC for circuit minimization. Demiurge is written in C++. The source code is available[1] under the GNU Lesser General Public License version 3.

In the following, we outline the architecture of Demiurge and then briefly explain the different back-ends.

II. Architecture

The architecture of Demiurge is outlined in Fig. 1. The input is a safety specification in AIGER format. The AIG2CNF module parses it into CNF formulas representing the transition relation and the set of safe states. Next, the back-end selected by the user is executed. The back-ends mostly differ in their method for computing the winning region, and can be parameterized with a method for computing the circuit from the winning region. Both the computation of the winning region and the extraction of circuits rely on external solvers like SAT- and QBF solvers. The resulting circuits are optimized with ABC and dumped in AIGER format again.

This work was supported in part by the Austrian Science Fund (FWF) through projects RiSE (S11406-N23 and S11408-N23) and QUAINT (I774-N23), and by the European Commission through project STANCE (317753).

[1]http://www.iaik.tugraz.at/content/research/design_verification/demiurge/

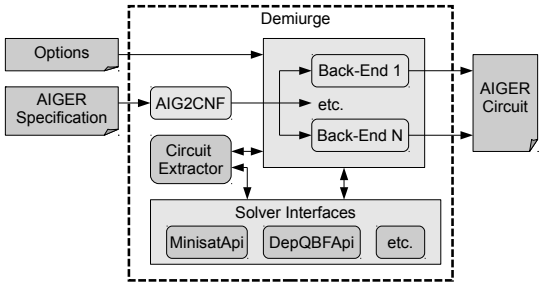

Fig. 1. The architecture of Demiurge.

III. Back-Ends

A. Learning-Based Back-End

The learning-based back-end computes a CNF representation of the winning region W in an iterative manner. It starts with the set of all safe states. In each iteration, it computes a state within the current version F of the winning region from which the environment can enforce to leave F. Obviously, such a state cannot be part of the final winning region W. Hence, the algorithm refines F by removing this state. The state is represented as a cube over the state variables, so removing it from F amounts to adding a clause. By dropping literals from the cube as long as it only contains states that must be excluded from the winning region, the algorithm generalizes the state into a larger region before removing it from the winning region. The detailed algorithm can be found in [1].

For SyntComp, we use the following configuration of this back-end. Instead of a QBF solver, we use two competing SAT solvers to compute and generalize states to be removed from the winning region (algorithm LearnSat from [1] with optimization RG enabled, but optimization RC disabled). Minisat version 2.2.0 is as underlying SAT solver (via its API, with default parameters).

B. Parallel Back-End

The parallel back-end is a playground for combining different methods that refine a CNF representation of the winning region iteratively with additional clauses. Several threads compute and add additional clauses in parallel.

For SyntComp, we use 3 threads. Two threads perform the same work as the learning-based back-end, one using Minisat version 2.2.0 and the other using Lingeling version ats. The third thread generalizes existing clauses of the winning region further by trying to drop more literals.

C. Template-Based Back-End

In order to obtain a winning region, this back-end constructs a parameterized CNF formula over the state variables: different concrete values for the (Boolean) parameters induce a different concrete CNF formula over the state variables. This way, the search for a formula over the state variables (the winning region) is reduced to a search for Boolean constants (the values of the template parameters). A QBF solver is finally used to compute values for the template parameters such that (a) the winning region contains only safe states, (b) the winning region contains the initial state, and (c) from each state of the winning region, the system player can enforce to reach a state of the winning region also in the next step.

For SyntComp, we use the following configuration of this back-end. DepQBF version 3.02 is used as QBF solver via its API. Bloqqer is used as QBF preprocessor. Bloqqer has been extended to preserve satisfying assignments [8].

D. Circuit Extraction

Demiurge provides two methods for computing circuits from the winning region. The first one uses QBFCert [7] to compute Skolem functions for the output signals in a QBF that asserts completeness of the strategy relation derived from the winning region. The second one uses computational as proposed in [3], but implemented with incremental SAT solving or incremental QBF solving instead of BDDs. A third method based on interpolation is under development.

For SyntComp, we use a CNF learning approach that computes circuits for one output after the other using a plain SAT solver. The approach is inspired by the substitution reduction method of [4] but with computational CNF learning [3] instead of interpolation. Lingeling ats is used as solver.

E. Other Back-Ends

Demiurge contains more back-ends that are either experimental or did not turn out to be particularly competitive. The *EPR back-end* [1] reduces the synthesis problem to Effectively Propositional Logic (EPR) and uses iProver to solve the formulas. It suffers from high memory consumption. The *IFM back-end* is a re-implementation of [6]. It performs well on certain benchmarks, but is outperformed on many others [1]. Demiurge also contains code for experimental back-ends trying to lift ideas from the model checking algorithm IC3 [2] to synthesis, as well as a QBF-based implementation of [6].

IV. Conclusion

Demiurge is an open-source synthesis tool for safety specifications. It implements several synthesis algorithms based on SAT- and QBF solving [1]. Demiurge is also an extendable framework for implementing new synthesis algorithms, thereby reducing the entry barrier for new research on SAT- and QBF-based synthesis algorithms and optimizations.

References

[1] R. Bloem, R. Könighofer, and M. Seidl. SAT-based synthesis methods for safety specs. In *VMCAI'14*. Springer, 2014.
[2] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI'11*, pages 70–87. Springer, 2011.
[3] R. Ehlers, R. Könighofer, and G. Hofferek. Symbolically synthesizing small circuits. In *FMCAD'12*. IEEE, 2012.
[4] J.-H. R. Jiang, H.-P. Lin, and W.-L. Hung. Interpolating functions from large boolean relations. In *ICCAD'09*. IEEE, 2009.
[5] F. Lonsing and U. Egly. Incremental QBF solving. *CoRR*, abs/1402.2410, 2014.
[6] A. Morgenstern, M. Gesell, and K. Schneider. Solving games using incremental induction. In *IFM'13*. Springer, 2013.
[7] A. Niemetz, M. Preiner, F. Lonsing, M. Seidl, and A. Biere. Resolution-based certificate extraction for QBF. In *SAT'12*. Springer, 2012.
[8] M. Seidl and R. Könighofer. Partial witnesses from preprocessed quantified boolean formulas. In *DATE'14*, pages 1–6. IEEE, 2014.