Journal on Satisfiability, Boolean Modeling and Computation 9 (2015) 53–58

Boolector 2.0

SYSTEM DESCRIPTION

Aina Niemetz aina.niemetz@jku.at
Mathias Preiner mathias.preiner@jku.at
Armin Biere biere@jku.at
Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Abstract

In this paper, we discuss the most important changes and new features introduced with version 2.0 of our SMT solver Boolector, which placed first in the QF_BV and QF_ABV tracks of the SMT competition 2014. We further outline some features and techniques that were not yet described in the context of Boolector.

Keywords: *SMT solving, Lemmas on Demand, Lambdas, Don't Care Reasoning*

Submitted November 2014; revised March 2015; published June 2015

1. Introduction

Boolector is a Satisfiability Modulo Theories (SMT) solver for the quantifier-free theories of fixed-size bit vectors and arrays. It employs the *lemmas on demand* approach, which is an extreme variant of lazy SMT. Recently, [10] introduced a generalization of the lemmas on demand procedure presented in [5] to lazily handle *lambda* terms. Further, an optimization of the lemmas on demand procedure based on don't care reasoning to reduce the cost for abstraction refinement was proposed in [9].

In this paper, we discuss our current version 2.0 of Boolector, which participated in tracks QF_BV and QF_ABV in the SMT competition 2014 and won both. We give an overview of the most important changes and new features since version 1.5.118, which won tracks QF_BV and QF_AUFBV of the SMT competition 2012.

2. Overview

Version 2.0 of Boolector implements the lemmas on demand for lambdas approach [10]. It further incorporates optimizations of the lemmas on demand procedure with don't care reasoning [9]. Figure 1 gives a high-level view of the procedure and introduces both the unoptimized approach LOD, and its optimized variant $\mathsf{LOD}_{\mathsf{opt}}$ as follows.

Given an input formula ϕ, procedure LOD enumerates truth assignments (candidate models) of the bit vector abstraction (bit vector skeleton) $\alpha(\pi)$ of the prepro-

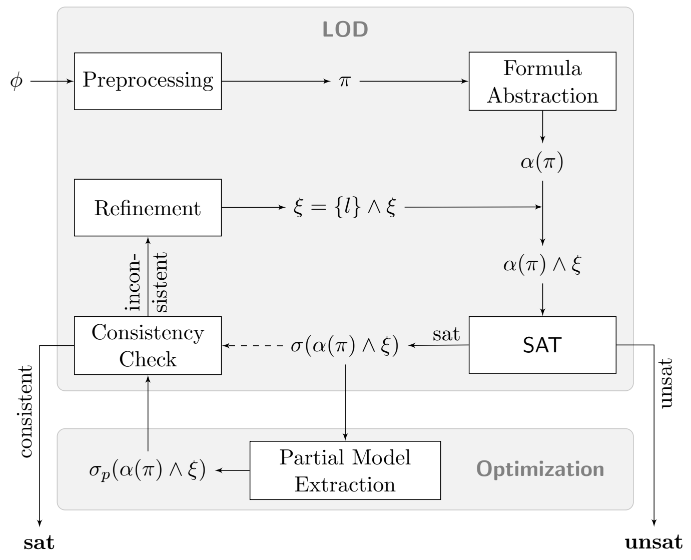

Figure 1. The workflow of the *lemmas on demand* decision procedure in Boolector. The original procedure LOD (indicated by the dashed line) works on full candidate models, whereas the optimized procedure $\mathsf{LOD_{opt}}$ extracts partial candidate models prior to consistency checking.

cessed input formula π and iteratively refines those assignments with lemmas until convergence. At refinement iteration i, formula refinement ξ is defined as $l_1 \wedge \ldots \wedge l_{i-1}$, where $l_1, \ldots, l_{i-1}$ is the set of lemmas derived up to iteration i. Note that initially, ξ is $\top$. In each iteration, an underlying decision procedure SAT determines the satisfiability of the (refined) formula abstraction $\Gamma \equiv \alpha(\pi) \wedge \xi$ by encoding Γ to SAT and determining its satisfiability by means of a SAT solver. As formula abstraction Γ is an overapproximation of ϕ, LOD immediately concludes with *unsat* if Γ is unsatisfiable. If Γ is satisfiable, the current (full) candidate model $\sigma(\alpha(\pi) \wedge \xi)$ is checked for consistency w.r.t. the preprocessed input formula π. If the candidate model is consistent, LOD immediately concludes with *sat*. Otherwise, the candidate model is *spurious* and a lemma l is added to the formula refinement ξ.

Abstraction refinement is usually the most costly part of LOD (requiring up to 99% of the total runtime), where cost correlates with the number of refinement iterations, as each iteration entails a call to the underlying SAT solver. Checking the full candidate model, however, is often not required, as only a small subset of the full model is responsible for actually satisfying the formula abstraction. Boolector 2.0 therefore implements an optimization of procedure LOD to exploit a posteriori observability don't cares, i.e., parts of the formula abstraction irrelevant to its satisfiability under the current assignment. By introducing don't care reasoning on full candidate models to extract partial candidate models prior to consistency checking, procedure $\mathsf{LOD_{opt}}$ focuses on the relevant parts of the formula abstraction only, and subsequently reduces the cost for consistency checking.

Note that the model checking approach for solving QF_BV formulas as proposed in [7] has not yet been implemented in Boolector 2.0.

2.1 Selected Features

Python API Since version 2.0, Boolector provides a public Python API as a Python module, which allows users to conveniently access the most important functions of Boolector's public C API from a Python program. The Python module makes use of Python operator overloading to create Boolector bit vector expressions and converts integer constants to Boolector constant expressions. It further maintains Boolector's reference counting for expressions and automatically releases allocated Boolector resources via the Python garbage collector. The documentation of Boolector's public API can be found at `http://fmv.jku.at/boolector/doc`.

Cloning Similar to cloning in Lingeling [2], Boolector 2.0 supports cloning and provides a *clone* function to generate an exact (but independent) copy of the original Boolector instance. Cloning is, e.g., extensively used in PBoolector [11], a parallel version of Boolector, which implements a cube and conquer approach similar to Treengeling [2], but for the quantifier-free theory of fixed-size bit vectors.

Model Generation Prior to version 2.0, Boolector with model generation enabled in some cases suffered from a performance drop. This was due to the fact that model generation required bit-blasting of terms that where previously eliminated during rewriting to generate assignments via the underlying SAT solver. Boolector 2.0 implements a new model generation algorithm, which determines assignments for such terms via term level propagation rather than via the SAT solver and therefore substantially reduces the previous overhead. Boolector provides two modes for generating models: construct a model 1) for all *asserted*, or 2) for all *constructed* expressions.

Internal Model Validation During testing, each time a formula is determined to be satisfiable, Boolector 2.0 internally checks if the resulting model is valid. This feature is enabled even if model generation is disabled. It uses cloning and the new model generation described above and helped to reveal several bugs in Boolector's rewriting engine. In the future, we will make this functionality available via Boolector's public API as well as the command (check-model) in SMT-LIB v2[1].

Uninterpreted Functions Boolector 2.0 natively supports uninterpreted functions (UF) with bit vector sorts. Note that prior to version 2.0, in SMT-LIB v1[1] Boolector was able to handle function applications on UF by means of reads on arrays, where the UF was interpreted as an array and the read index was a concatenation of the function application's arguments.

Skeleton Preprocessing As a preprocessing step, Boolector attempts simplification of the Boolean structure of the formula by means of SAT preprocessing. It generates a Boolean formula abstraction (Boolean skeleton), which is encoded to SAT

1. `http://www.smtlib.org`

and simplified by the preprocessor of the underlying SAT solver. In case that the SAT solver returns *unsatisfiable*, Boolector immediately concludes with *unsatisfiable*. For the *satisfiable* case, however, Boolector extracts all unit clauses found by the SAT solver, maps them back to the term level, and adds them as new constraints, which in turn may be used for further simplifications of the formula.

Unconstrained Optimization Boolector 2.0 reintroduces a previously disabled optimization based on so-called unconstrained variables [3], where unconstrained terms are substituted by fresh variables to simplify the formula. Boolector's unconstrained optimization now also considers lambdas and UF. Note that model generation with unconstrained optimization enabled is not yet supported.

Symbolic Lemmas In contrast to previous versions, Boolector 2.0 now maintains lemmas on the term level rather than directly encoding them to CNF. This enables lemma sharing between Boolector instances (as employed, e.g., in our dual propagation-based optimization [9]).

Model-based Testing We use model-based testing as in [1] to extensively test Boolector's public C API. This is a very effective and efficient automated testing approach to test all solver features exposed to the user. Prior to model-based testing, we applied grammar-based black-box input fuzzing in combination with delta debugging [4, 8]. In contrast to model-based testing, however, the effectiveness of input fuzzing is restricted by the input format's expressiveness. Consequently, if the solver supports features not supported by the format, these features cannot be tested via input fuzzing unless the format (and its respective parser) is extended to support them. Note that we still rely on input fuzzing for testing Boolector's parsers[2].

API Tracing Every call to Boolector's public API can be recorded via API tracing. The resulting trace can be replayed and the replayed sequence behaves exactly like the original Boolector run. This is particularly useful for debugging purposes, as it enables replaying erroneous behaviour.

Improvements of the Incremental API We improved Boolector's assumption handling, which now also provides means to identify failed assumptions [6].

3. Lemmas on Demand for Lambdas

Boolector implements a new lemmas on demand decision procedure for lambda terms as introduced in [10], which replaces the array decision procedure introduced in [5]. Internally, Boolector now represents array variables as uninterpreted functions (UF), array operations *write* and *if-then-else* as lambda terms, and *read* operations as function applications. Lambda terms enable the modeling of array operations other than the base operations introduced above, e.g., constant, pre-initialized arrays, or *memcpy* and *memset* from the standard C library. In addition to modelling arrays, lambdas in Boolector can also be used to create arbitrary functions, with the only restriction

2. http://fmv.jku.at/ddsexpr

that those functions may neither be recursive nor of higher order. Boolector's new lemmas on demand procedure for lambdas is able to lazily handle lambda terms, i.e., lambdas are instantiated on demand during consistency checking. Further, Boolector optionally supports the eager elimination of lambda terms, which may—in the worst case—result in an exponential blow-up in the size of the formula. However, on certain instances—especially in the field of software model checking and symbolic execution—eager elimination of lambda terms is beneficial and yields significant simplifications through rewriting. Note that Boolector 2.0 does not yet support equality over lambda terms and consequently does not yet support extensionality on arrays.

4. Don't Care Reasoning on the Bit Vector Skeleton

As introduced in [9] and indicated in Fig. 1, Boolector 2.0 implements an optimization of the lemmas on demand procedure in [10] to reduce the cost for abstraction refinement. It implements two techniques for extracting *partial candidate models* by identifying irrelevant parts of the formula abstraction via don't care reasoning prior to consistency checking. This subsequently reduces the number of refinement iterations and consequently, the overall runtime of the lemmas on demand procedure.

Our *justification*-based approach identifies a posteriori observability don't cares by skipping lines that do not influence the output of an AND gate under the current assignment of the formula abstraction based on the fact that Boolector internally represents a formula as directed acyclic graph (DAG), where all Boolean expressions are expressed by means of NOT and (two-input) AND gates.

Our *dual propagation*-based approach, on the other hand, exploits the duality of the formula abstraction, i.e., the fact that assignments satisfying Γ (the *primal* channel) falsify its negation $\neg\Gamma$ (the *dual* channel). We follow an *offline* strategy with *one* solver *per* channel (rather than an *online* strategy with one solver for *both* channels), where the primal solver generates a full assignment before the dual solver enables partial model extraction based on the primal assignment.

5. SMT Competition 2014 Configuration

Three different configurations of Boolector participated in two tracks at the SMT competition 2014. Configuration (1) Boolector entered the QF_BV track with *unconstrained optimization* and *full beta reduction* (to eagerly eliminate SMT-LIB v2 macros) enabled. The other two configurations, (2) Boolector (dual propagation) and (3) Boolector (justification) entered the QF_ABV track with *dual propagation* resp. *justification* optimization, *unconstrained optimization*, and *full beta reduction probing* enabled. Boolector 2.0 is a cleaned-up version of the version that participated in the SMT competition 2014. It is configured to reenact the above configurations as follows: (1) `-bra -uc`, (2) `-pbra -uc -dp`, and (3) `-pbra -uc -ju`. Note that since Boolector 2.0 does not yet support extensionality, for benchmarks that were still extensional after rewriting (174 out of 6457), we used version 1.5.118 of Boolector (with the SAT competition 2014 version azd of Lingeling), which implements the old lemmas on demand engine and therefore still supports extensionality on arrays.

6. Conclusion

We presented Boolector 2.0, a new version of our SMT solver Boolector, and discussed the most important changes and new features introduced since Boolector version 1.5.118. We outlined techniques like *skeleton preprocessing* and *cloning* that have not yet been described in the context of Boolector, and discussed *model-based testing*, which helped us tremendously to improve and implement new features in Boolector.

References

[1] Cyrille Artho, Armin Biere, and Martina Seidl. Model-Based Testing for Verification Back-Ends. In *TAP*, **7942** of *LNCS*. Springer, 2013.

[2] A. Biere. Yet Another Local Search Solver and Lingeling and Friends Entering the SAT Competition 2014. In *SAT Competition 2014*, Univ. of Helsinki, 2014.

[3] Robert Brummayer. *Efficient SMT Solving for the Extensional Theory of Arrays.* PhD thesis, JKU Linz, 2009.

[4] Robert Brummayer and Armin Biere. Fuzzing and Delta-Debugging SMT solvers. In *SMT 2009*. ACM, 2009.

[5] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, **6**(1-3), 2009.

[6] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT'04*, **2919** of *LNCS*. Springer, 2004.

[7] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. Efficiently solving bit-vector problems using model checkers. In *SMT 2013*, Helsinki, Finland, 2013.

[8] Aina Niemetz and Armin Biere. ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In *SMT 2014*, Helsinki, Finland, 2013.

[9] Aina Niemetz, Mathias Preiner, and Armin Biere. Turbo-charging lemmas on demand with don't care reasoning. In *FMCAD'14*. IEEE, 2014.

[10] Mathias Preiner, Aina Niemetz, and Armin Biere. Lemmas on demand for lambdas. In *DIFTS'13*, **1130** of *CEUR Workshop Proceedings*, 2013.

[11] Christian Reisenberger. PBoolector: A Parallel SMT Solver for QF_BV by Combining Bit-Blasting and Look-Ahead. Master's thesis, JKU Linz, 2014.