AProVE at SMT-COMP 2020

Marc Brockschmidt[1], Florian Frohn[2], Carsten Fuhs[3], Jürgen Giesl[4], Jera Hensel[4], Peter Schneider-Kamp[5], Thomas Ströder[6], and René Thiemann[7]

[1] Microsoft Research Cambridge, United Kingdom
[2] AbsInt GmbH, Germany
[3] Birkbeck, University of London, United Kingdom
[4] RWTH Aachen University, Germany
[5] University of Southern Denmark, Denmark
[6] IT-P Information Technology-Partner GmbH, Germany
[7] University of Innsbruck, Austria

In automated tools for termination proving and complexity bound inference like our tool AProVE [4], often the need arises to solve Boolean combinations of constraints in non-linear (integer) arithmetic to perform a proof step for a successful termination proof. Examples for prominent termination proof techniques where this is the case are well-founded orders based on polynomial [3] or matrix interpretations [2]. In order to facilitate this task, AProVE features a dedicated SMT solver for the SMT-LIB logic QF_NIA. AProVE is written in Java.

The approach we are using at SMT-COMP for the QF_NIA category is based on a reduction to satisfiability problems on finite domains for the unknowns. In case a satisfying assignment is found by the finite domain solver, we return the corresponding integer solution. If the finite domain solver detects unsatisfiability of the generated instance for the current search space, we know that there is no solution for the QF_NIA instance with the used finite domain, and we restart the search with an extended domain. In this way, we obtain a semi-decision procedure for the satisfiability problem of QF_NIA. We additionally use information from global constraints (i.e., atomic top-level assertions like $x \geqslant 42$) of the QF_NIA instance to bound the search space for certain unknowns.

To solve the generated instances of finite-domain satisfiability problems, AProVE uses an encoding to the satisfiability problem of propositional logic (SAT). We first generate a propositional formula DAG (which shares common subexpressions; this approach is also known as *structural hashing*) and then convert this DAG into an equisatisfiable conjunctive normal form (CNF) using the implementation of Tseitin's transformation in SAT4J [5]. This CNF is then checked for satisfiability by the SAT solver MiniSat [1]. This kind of approach is commonly known as *bit-blasting*.

Our SAT encoding is described in detail in [3]. At a high level, it works as follows. The Boolean structure of the original constraint is represented as such. It thus remains to encode atomic constraints from QF_NIA. To this end, we assign a bitvector of Boolean variables to each of the unknowns. These bitvectors are large enough for a binary representation of all values of the chosen finite domain. We then encode the corresponding arithmetic operations in such a way that the bitvector for the result is again large enough to store the result of the operation. For example, if the unknowns x and y are represented in m bits, the encoding of $x + y$ will use $m + 1$ bits.

This is in contrast to the logic QF_BV, where the length m of the bitvector for the result of an arithmetic operation like addition or multiplication is the same as for its inputs, such that computations expressed in QF_BV are modulo 2^m (see also http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_BV).

We also keep track of the maximum value that an expression can take based on the search space of its components, which allows us to drop most significant bits if we detect that they will necessarily be equivalent to 0. As an example, consider a product $x \cdot y \cdot z$, where we search for solutions for x, y, z over $\{0, 1, 2, 3\}$. Thus, each of x, y, z is represented by a bitvector of 2 bits. Here the SAT encoding for multiplication would yield a bitvector of $2 + 2 + 2 = 6$ bits. However, the maximum possible value for the expression is $3 \cdot 3 \cdot 3 = 27$, which uses only 5 bits in binary representation. Thus, we can drop the most significant bit from the bitvector for the product $x \cdot y \cdot z$ since it will always be equivalent to 0.

For the structural hashing, equality of arguments for $\vee$ and $\wedge$ is considered modulo associativity, commutativity, and multiplicity, i.e., we represent both $(x \vee y) \vee x$ and $y \vee (y \vee x)$ by $x \vee y$. Similarly, for the exclusive-or $\oplus$ we use that $x \oplus x$ is equivalent to 0. Finally, we use partial evaluation of formulas involving Boolean constants during construction (e.g., when we create a disjunction of 0 and a formula p, we obtain p instead of $0 \vee p$).

Acknowledgments. We thank Karsten Behrmann, Andrej Dyck, Fabian Emmes, and Patrick Kabasci for their contributions to the SMT-solving front-end of AProVE.

References

[1] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004. See also http://minisat.se.

[2] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[3] C. Fuhs, J. Giesl, A. Middeldorp, R. Thiemann, P. Schneider-Kamp, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT 2007*, volume 4501 of *LNCS*, pages 340–354, 2007.

[4] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58:3–31, 2017. AProVE project URL: http://aprove.informatik.rwth-aachen.de/. Binary for SMT-COMP 2020 available at: https://www.starexec.org/starexec/secure/details/solver.jsp?id=1229.

[5] D. Le Berre and A. Parrain. The SAT4J library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. See also http://www.sat4j.org.