Ultimate TreeAutomizer
(CHC-COMP Tool Description)

Daniel Dietsch
University of Freiburg
dietsch@cs.uni-freiburg.de

Matthias Heizmann
University of Freiburg
heizmann@cs.uni-freiburg.de

Jochen Hoenicke
University of Freiburg
hoenicke@cs.uni-freiburg.de

Alexander Nutz
University of Freiburg
nutz@cs.uni-freiburg.de

Andreas Podelski
University of Freiburg
podelski@cs.uni-freiburg.de

We present Ultimate TreeAutomizer, a solver for satisfiability of sets of constrained Horn clauses. Constrained Horn clauses (CHC) are a fragment of first order logic with attractive properties in terms of expressiveness and accessibility to algorithmic solving. Ultimate TreeAutomizer is based on the techniques of trace abstraction, tree automata and tree interpolation. This paper serves as a tool description for TreeAutomizer in CHC-COMP 2019.

1 Introduction

We present Ultimate TreeAutomizer, a solver for satisfiability of sets of constrained Horn clauses. The logical fragment of constrained Horn clauses (CHC) has received increasing attention in the last years. One reason for its attractiveness in program verification is that it naturally allows for expressing proof queries for many kinds of correctness proofs, e.g., classic Floyd-Hoare proofs for while-programs, but also assume-guarantee reasoning, compositional verification, and many more [11, 15].

The CHC fragment is equivalent in expressive power to the verification of safety properties of procedural (possibly recursive) programs, i.e., there is a translation of a CHC-formula to a procedural program such that the CHC-formula is satisfiable if and only if the procedural program is correct, and vice versa. Therefore, it is not surprising that solvers for CHC-formulas often adapt algorithms known in program verification. For example, HSF [10, 3] uses predicate abstraction, Spacer[1] uses PDR [9, 14], and Rahft [17] uses trace abstraction [12], to name just a few tools. Ultimate TreeAutomizer is part of this tradition and is an adaptation of the trace abstraction verification algorithm for procedural programs [13].

This paper is a tool description for the TreeAutomizer tool as it participated in CHC-COMP in 2018 and 2019. We give a brief overview of how trace abstraction is used to solve CHC-formulas. Afterwards, we describe some aspects of the implementation of TreeAutomizer and some crucial optimizations. Last, we discuss expected strengths and weaknesses of the approach.

2 Approach

In this section, we describe the approach for solving formulas in the CHC-fragment used in TreeAutomizer. The approach is based on trace abstraction [12]; its adaptation to solving CHC-formulas has

[1] https://spacer.bitbucket.io

been described by Kafle and Gallagher [16] and by Wang and Jiao [19], we refer to these papers for a more in-depth description and only give an overview here.

In the following, we assume that a constraint theory T is given, and that we have an SMT-solver for T. Furthermore, we refer to constraints over theory T with free variables $\vec{x}$ as $C(\vec{x})$ and we assume that a set $\{P_1, P_2, \ldots\}$ of predicate symbols is given that are not used by the constraint theory T.

A formula in the CHC-fragment is given as a set of clauses where each clause is of one of the below forms. According to general convention, Horn clauses subdivided the categories of *facts*, *definite clauses*, and *queries* (also: *goal clauses*), depending on which of the below patterns they match.

$$\begin{aligned} \forall \vec{x}.\, C(\vec{x}) &\rightarrow P(\vec{x}) && \text{(fact)} \\ \forall \vec{x}.\, P_1(\vec{x}) \land \ldots \land P_n(\vec{x}) \land C(\vec{x}) &\rightarrow P(\vec{x}) && \text{(definite clause)} \\ \forall \vec{x}.\, P_1(\vec{x}) \land \ldots \land P_n(\vec{x}) \land C(\vec{x}) &\rightarrow \mathsf{false} && \text{(query)} \end{aligned}$$

In the remainder, we assume that a set S of constrained Horn clauses is given.

Now, let us consider the resolution trees over the clauses in set S with root false. We call such a tree a *derivation of* false. Since no constraints ever occur in a clause head, the resolvent at the root of a derivation of false is a query with one large conjunctive constraint in the antecedent, i.e., it is of the following form.

$$\forall \vec{x}.\, C_1(\vec{x}) \land \ldots \land C_n(\vec{x}) \rightarrow \mathsf{false}$$

We call a derivation of false *feasible* if the formula $\exists \vec{x}.\, C_1(\vec{x}) \land \ldots \land C_n(\vec{x})$ is satisfiable, and *infeasible* otherwise. The existence of a feasible derivation of false means that the conjunction of the clauses in S is contradictory. Completeness of first-order resolution implies that the converse also holds, i.e., that the absence of a feasible derivation of false implies satisfiability of the formula. Thus, we can formulate the following proof rule.

> A set of constraint Horn clauses S is satisfiable if and only if there is no feasible derivation of false over S.

Ultimate TreeAutomizer's approach to prove satisfiability of the set of Horn clauses S is to show infeasibility of all derivations of false over S. The refinement algorithm used for this purpose is shown in Figure 1. The proving process starts by sampling a derivation from the set of all derivations of false over S. The sample derivation is then checked for feasibility using an SMT solver. If the sample derivation is feasible, the clause set S is unsatisfiable (since it implies false). If the sample derivation is infeasible, the sample is generalized to a set of derivations which are all infeasible. This set is subtracted from the set of derivations of false. This process is repeated until either all derivations of false have been proven infeasible or a feasible derivation has been found.

3 Implementation

TreeAutomizer is implemented in the Ultimate framework. It is written in Java, open source, and can be downloaded and contributed to on Ultimate's Github page[2].

Ultimate provides for TreeAutomizer the SMTLIB parser, utilities for handling formulas (e.g., simplifications), and the Ultimate Automata Library. SMT solvers for which by Ultimate provides an interface include SMTInterpol [4], Z3 [18], CVC4 [2], and MathSat [5]. In cases when a solver does not support interpolation in the given constraint theory, but can produce unsatisfiable cores, Newton-style interpolation [8] can be used to obtain interpolants.

TreeAutomizer takes as input Horn clause sets in the format used in CHC-COMP[3]. During parsing, the input formulas are converted into the normal form given above.

In order to represent (possibly infinite) sets of derivations of false, TreeAutomizer uses tree automata (see [7] for more details on tree automata). The alphabet that the tree automata operate on is the set of input Horn clauses S. The states of the tree automata have one of two different semantics. The states of the automata A, and A_S represent the uninterpreted predicates in the set $\{P_1, P_2 \ldots\}$. The states of the interpolant automaton G represent the interpolants from the interpolation query that is generated from the sample derivation of false d. From this sample query, a generalization procedure computes the canonical interpolant automaton. The canonical interpolant automaton is given by the set of all rules that correspond to a valid implication between the formulas in the source of the automaton rules, the constraint in the alphabet symbol, and the formula at target of the rule (see [19]) for a thorough description).

```
A := A_S
while (nonempty(A)) {
  d := sample(A)
  if (d is feasible)
    return unsat
  I := getTreeInterpolant(d)
  G := generalize(d, I, S)
  A := A \ G
}
return sat
```

Figure 1: Trace abstraction refinement scheme used in Ultimate TreeAutomizer. S is the input set of constrained Horn clauses. A_S is a set containing all derivations of false over S. The procedure sample picks an element from a non-empty set. The procedure generalize takes an infeasible derivation of false as input and returns a set of infeasible derivations of false that contains at least the input derivation (see also Figure 2). Note that the check for feasibility as well as the generalize procedure rely on calls to an (interpolating) SMT solver.

4 Optimizations

In each iteration of the main refinement loop of trace abstraction, an interpolant automaton (G) is created and subtracted from the automaton representing the derivations of false (A). Two major bottlenecks in terms of space and time consumption may arise from this. First, the generalization that is done during creation of the interpolant automaton can produce a large number of candidate transition rules each one requiring an SMT solver call. Second, the difference operation requires construction of a product automaton and thus can lead to growth of the automaton representing the derivations of false that is exponential in the number of loop iterations. Both problems are amplified by an increasing nonlinearity of the involved Horn clauses.

[2]https://github.com/ultimate-pa/

[3]https://chc-comp.github.io/2018/format.html

Minimization The explosion through the repeated product construction can often be contained through an additional minimization step on the result of the difference operation. Standard minimization algorithms for tree automata can be used here; the Ultimate automata library currently supports two minimization variants, one based on the naive algorithm [7] the other on bisimulation [1].

On-Demand Construction of Interpolant Automaton The explosion of the number of rules in the interpolant automaton can be countered by integrating the difference (i.e., complementation and intersection) operation with the creation of the interpolant automaton.

The idea behind the integration is that a large number of candidate rules in the interpolant automaton is irrelevant to the result of the difference operation. The basic intuition here is that for computing the difference $S \setminus T$ of two sets S and T, only the part of T that lies in the intersection of S and T is relevant – elements of T that don't lie in S need not be considered by a subtraction algorithm. For the subtraction of tree automata, this means the following: A rule is irrelevant to the result of the difference operation, `A` $-$ `G`, if it never contributes to the construction of a tree in `G` that lies in the language of `A`. Such candidate rules can be filtered out during the product construction by only querying the interpolant automaton for rules whose source tuple is reachable in according to the minuend automaton (`A`).

```
generalize(d, I, S) {
  result = freshTreeAutomaton(S);
  for ((P_1 ... P_n /\ C -> P) in S) {
    for (formulas (phi_1, ..., phi_(n+1))
       with phi_i occur in I) {
      candidateRule :=
        phi_1 /\ ... phi_n /\ c -> phi_n+1;
      if (checkvalidity(candidateRule))
        result.addRule(candidateRule);
    }
  }
  return result;
}
```

Figure 2: `generalize` procedure as called in the refinement algorithm in Figure 1. Given an infeasible derivation of `false` `d`, a tree interpolant *I*, and the Horn clause set `S`, the procedure returns a tree automaton that accepts `d` and, by generalization, possibly other infeasible derivations of `false`. The tree automaton's states are the predicates that occur in the tree interpolant `I`. The generalization happens through adding rules to the tree automaton that correspond to a valid implication between the source states, conjoined with the constraint in the alphabet symbol (a Horn clause from *S*), and the target state.

5 Discussion

TreeAutomizer's approach inherits its basic properties from the trace abstraction approach. Thus, TreeAutomizer is conceptually sound and relatively complete. As is common for such refinement schemes, the actual detection of a proof of satisfiability (and thus actual completeness) depends on guessing the right formulas during the generalization step (we only mentioned interpolation here, but several other methods are available).

We believe that one strength of the trace abstraction approach lies in a semantic independence of refinement steps. For example in predicate abstraction with CEGAR [6] (which several program verification schemes can be seen as a variant of), formulas that stem from many different refinement steps are conjoined. This means that SMT-solver queries get bigger and bigger over a growing number of iterations, which can swamp the SMT solver. In trace abstraction on the other hand, the formulas used in the `generalize` procedure can be forgotten, after the difference `A - G` has been computed, i.e. formulas from different refinement steps are never conjoined. However, among other things, this property relies on a rich-enough structure of the initial automaton. In particular, this means that CHC-formulas that stem from proof queries for programs where large block encoding has been performed or where the program counter is not made explicit by using different uninterpreted predicates for each location, this compositionality may not come into full effect.

References

[1] Parosh Aziz Abdulla, Lisa Kaati & Johanna Högberg (2006): *Bisimulation Minimization of Tree Automata*. In: *CIAA*, *Lecture Notes in Computer Science* 4094, Springer, pp. 173–185, doi:10.1137/0216062.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In: *CAV*, *Lecture Notes in Computer Science* 6806, Springer, pp. 171–177, doi:10.1007/3-540-45657-0_40.

[3] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In: *Fields of Logic and Computation II*, *Lecture Notes in Computer Science* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-19249-9.

[4] Jürgen Christ, Jochen Hoenicke & Alexander Nutz (2012): *SMTInterpol: An Interpolating SMT Solver*. In: *SPIN*, *Lecture Notes in Computer Science* 7385, Springer, pp. 248–254, doi:10.1007/11532231_26.

[5] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma & Roberto Sebastiani (2013): *The MathSAT5 SMT Solver*. In: *TACAS*, *Lecture Notes in Computer Science* 7795, Springer, pp. 93–107, doi:10.1007/978-3-642-31365-3_38.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu & Helmut Veith (2000): *Counterexample-Guided Abstraction Refinement*. In: *CAV*, *Lecture Notes in Computer Science* 1855, Springer, pp. 154–169, doi:10.1007/3-540-49519-3_18.

[7] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications*. Available on: http://www.grappa.univ-lille3.fr/tata. Release October, 12th 2007.

[8] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz & Andreas Podelski (2017): *Craig vs. Newton in software model checking*. In: *ESEC/SIGSOFT FSE*, ACM, pp. 487–497, doi:10.1145/3106237.3106307.

[9] Niklas Eén, Alan Mishchenko & Robert K. Brayton (2011): *Efficient implementation of property directed reachability*. In: *FMCAD*, FMCAD Inc., pp. 125–134.

[10] Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *HSF(C): A Software Verifier Based on Horn Clauses - (Competition Contribution)*. In: *TACAS*, *Lecture Notes in Computer Science* 7214, Springer, pp. 549–551, doi:10.1007/978-3-540-69611-7_16.

[11] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *PLDI*, ACM, pp. 405–416, doi:10.1145/2254064.2254112.

[12] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2009): *Refinement of Trace Abstraction*. In: *SAS*, *Lecture Notes in Computer Science* 5673, Springer, pp. 69–85, doi:10.1007/978-3-540-73368-3_46.

[13] Matthias Heizmann, Jochen Hoenicke & Andreas Podelski (2010): *Nested interpolants*. In: *POPL*, ACM, pp. 471–482, doi:10.1145/1706299.1706353.

[14] Krystof Hoder & Nikolaj Bjørner (2012): *Generalized Property Directed Reachability*. In: *SAT*, *Lecture Notes in Computer Science* 7317, Springer, pp. 157–171, doi:10.1007/3-540-49481-2_26.

[15] Jochen Hoenicke, Rupak Majumdar & Andreas Podelski (2017): *Thread modularity at many levels: a pearl in compositional verification*. In: *POPL*, ACM, pp. 473–485, doi:10.1145/3009837.3009893.

[16] Bishoksan Kafle & John P. Gallagher (2015): *Tree Automata-Based Refinement with Application to Horn Clause Verification*. In: *VMCAI*, *Lecture Notes in Computer Science* 8931, Springer, pp. 209–226, doi:10.1007/3-540-52753-2_52.

[17] Bishoksan Kafle, John P. Gallagher & José F. Morales (2016): *Rahft: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata*. In: *CAV (1)*, *Lecture Notes in Computer Science* 9779, Springer, pp. 261–268, doi:10.1016/j.jsc.2010.06.005.

[18] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *TACAS*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340, doi:10.1109/MS.2006.117.

[19] Weifeng Wang & Li Jiao (2016): *Trace Abstraction Refinement for Solving Horn Clauses*. *Comput. J.* 59(8), pp. 1236–1251, doi:10.1145/69575.69577.