Science of Computer Programming 111 (2015) 3–22

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Theory propagation and reification

Ed Robbins [a,*], Jacob M. Howe [b], Andy King [a]

[a] School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK
[b] Department of Computer Science, City University London, London, EC1V 0HB, UK

ARTICLE INFO

Article history:
Received 11 December 2013
Received in revised form 22 April 2014
Accepted 19 May 2014
Available online 2 June 2014

Keywords:
SMT
SAT
Satisfiability Modulo Theories
Reification
DPLL
Rational tree unification
Type recovery
Reverse engineering
Difference logic

ABSTRACT

SAT Modulo Theories (SMT) is the problem of determining the satisfiability of a formula in which constraints, drawn from a given constraint theory T, are composed with logical connectives. The DPLL(T) approach to SMT has risen to prominence as a technique for solving these quantifier-free problems. The key idea in DPLL(T) is to couple unit propagation in the propositional part of the problem with theory propagation in the constraint component. In this paper it is demonstrated how reification provides a natural way for orchestrating this in the setting of logic programming. This allows an elegant implementation of DPLL(T) solvers in Prolog. The work is motivated by a problem in reverse engineering, that of type recovery from binaries. The solution to this problem requires an SMT solver where the theory is that of rational-tree constraints, a theory not supported in off-the-shelf SMT solvers, but realised as unification in Prolog systems. The approach is also illustrated with SMT solvers for linear constraints and integer difference constraints. The rational-tree solver is benchmarked against a number of type recovery problems, and compared against a lazy-basic SMT solver built on PicoSAT, while the integer difference logic solver is benchmarked against CVC3 and CVC4, both of which are implemented in C++.

1. Introduction

DPLL-based SAT solvers have advanced to the point where they can rapidly decide the satisfiability of structured problems that involve thousands of variables. SAT Modulo Theories (SMT) seek to extend these ideas beyond propositional formulae to formulae that are constructed from logical connectives that combine constraints drawn from a given underlying theory. This section introduces the motivating problem of type recovery and explains why it leads to work on theory propagation in a Prolog SMT solver.

1.1. Type recovery with SMT

The current work is motivated by reverse engineering and the problem of type recovery from binaries. Reversing executable code is of increasing relevance for a range of applications:

- Exposing flaws and vulnerabilities in commercial software, especially prior to deployment in government or industry [13,19];

* Corresponding author.
E-mail address: er209@kent.ac.uk (E. Robbins).

http://dx.doi.org/10.1016/j.scico.2014.05.013
0167-6423/© 2014 Published by Elsevier B.V.

- Reuse of legacy software without source code for guaranteed compliance with hardware IO or timing behaviour, for example, for hardware drivers [11] or control systems [8];
- Understanding the operation of, and threat posed by, viruses and other malicious code by anti-virus companies [50].

An important problem in reverse engineering is that of type recovery [43]. A fragment of binary code will almost certainly have multiple source code equivalents, will contain a variety of complex addressing schemes, and during compilation will have lost most, if not all, of the type information explicit in the original source code. Additionally, container-like entities, analogous to high level source code variables and objects, cannot be readily extracted from binary code. The recovery of variables and their types is an essential component of reverse engineering, which makes understanding the semantics of the program considerably easier.

This paper observes that type recovery can be formulated as an SMT problem over rational-trees, a theory that in the context of type checking is referred to as circular unification [38]. Circular unification allows recursive types to be discovered in which a type variable can be unified with a term containing it. The use of rational-trees for type inference is not a new idea [38], but its application to the recovery of recursive types from an executable is far from straightforward because each instruction can be assigned many different types. Many SMT solvers include the theory of equality logic over uninterpreted functors [31,45] which is strictly weaker than circular unification and cannot capture recursive types. Unfortunately the theory of rational-trees is not currently supported in any off-the-shelf SMT solver, hence this investigation into how to build a solver.

1.2. SMT solving with lazy-basic

One straightforward approach to SMT solving is to apply the so-called lazy-basic technique which decouples SAT solving from theory solving. To illustrate, consider the SMT formula $f = (x \leq -1 \vee -x \leq -1) \wedge (y \leq -1 \vee -y \leq -1)$ and the SAT formula $g = (p \vee q) \wedge (r \vee s)$ that corresponds to its propositional skeleton. In the skeleton, the propositional variables p, q, r and s, respectively, indicate whether the theory constraints $(x \leq -1)$, $(-x \leq -1)$, $(y \leq -1)$ and $(-y \leq -1)$ hold. In this approach, a model is found for $(p \vee q) \wedge (r \vee s)$, for instance, $\{p \mapsto \mathit{true}, q \mapsto \mathit{true}, r \mapsto \mathit{true}, s \mapsto \mathit{false}\}$. Then, from the model, a conjunction of theory constraints $(x \leq -1) \wedge (-x \leq -1) \wedge (y \leq -1) \wedge \neg(-y \leq -1)$ is constructed, with the polarity of the constraints reflecting the truth assignment. This conjunction is then tested for satisfiability in the theory component. In this case it is unsatisfiable, which triggers a diagnostic stage. This amounts to finding a conjunct, in this case $(x \leq -1) \wedge (-x \leq -1)$, which is also unsatisfiable, that identifies a source of the inconsistency. From this conjunct, a blocking clause $(\neg p \vee \neg q)$ is added to g to give g' which ensures that conflict between the theory constraints is never encountered again. Then, solving the augmented propositional formula g' might, for example, yield the model $\{p \mapsto \mathit{false}, q \mapsto \mathit{true}, r \mapsto \mathit{true}, s \mapsto \mathit{true}\}$, from which a second clause $(\neg r \vee \neg s)$ is added to g'. Any model subsequently found, for instance, $\{p \mapsto \mathit{false}, q \mapsto \mathit{true}, r \mapsto \mathit{true}, s \mapsto \mathit{false}\}$, will give a conjunction that is satisfiable in the theory component, thereby solving the SMT problem.

The lazy-basic approach is particularly attractive when combining an existing SAT solver with an existing decision procedure, for instance, a solver provided by a constraint library. By using a foreign language interface a SAT solver can be invoked from Prolog [12] and a constraint library can be used to check satisfiability of the conjunction of theory constraints. A layer of code can then be added to diagnose the source of any inconsistency. This provides a simple way to construct an SMT solver that compares very favourably with the coding effort required to integrate a new theory into an existing open source SMT solver. The latter is normally a major undertaking and often can only be achieved in conjunction with the expert who is responsible for maintaining the solver. Furthermore, few open source solvers are actively maintained. Thus, although one might expect implementing a new theory to be merely an engineering task, it is actually far from straightforward.

Prolog has rich support for implementing decision procedures for theories, for instance, attributed variables [20,21]. (Attributed variables provide an interface between Prolog and a constraint solver by permitting logical variables to be associated with state, for instance, the range of values that a variable can possibly assume.) Several theories come prepackaged with many Prolog systems. This raises the questions of how to best integrate a theory solver with a SAT solver, and how powerful an SMT solver written in a declarative language can actually be. This motivates further study of the coupling between the theory and the propositional component of the SAT solver which goes beyond the lazy-basic approach, to the roots of logic programming itself.

The equation Algorithm = Logic + Control [33] expresses the idea that in logic programming algorithm design can be decoupled into two separate steps: specifying the logic of the problem, classically as Horn clauses, and orchestrating control of the sub-goals. The problem of satisfying a SAT formula is conceptually one of synchronising activity between a collection of processes where each process checks the satisfiability of a single clause. Therefore it is perhaps no surprise that control primitives such as delay declarations [44] can be used to succinctly specify the watched literal technique [42]. In this technique, a process is set up to monitor two variables of each clause. To illustrate, consider the clause $(x \vee y \vee \neg z)$. The process for this clause will suspend on two of its variables, say x and y, until one of them is bound to a truth-value. Suppose x is bound. If x is bound to *true* then the clause is satisfied, and the process terminates; if x is bound to *false*, then the process suspends until either y or z is bound. Suppose z is subsequently bound, either by another process or by labelling. If z is *true* then y is bound to *true* since otherwise the clause is not satisfied; if z is *false* then the clause is satisfied and the process closes down without inferring any value for y. Note that in these steps the process only waits on two variables at any one time. Unit propagation is at the heart of SAT solving and when implemented by watched literals