project.sty
1

COMPILATION OF A CSP LIKE LANGUAGE

MATHIAS SSERWADDA 215003597

April 19, 2017

1 First section

1 Introduction

This compiler has been developed with the aim of building a system for easy CSP modeling and solving. By taking advantage of the year-over-year increase in performance of SMT solvers, we hope that such a system can serve as an alternative to other decision procedures in many applications. The compiler can also be used for easy SMT benchmark generation.

2 Background to The Problem

Over the last decade there have been important advances in logic based techniques and tools. Advances have been especially significant in the field of propositional satisfiability (SAT), to the point that nowadays modern SAT solvers can tackle real-world problem instances with millions of variables. Hence, SAT solvers have become a viable engine for solving combinatorial discrete problems. For instance, an application that compiles specifications written in a declarative modeling language into SAT is shown to give promising results. For some applications of SAT technology on industrial problems. Interesting comparisons between SAT and Constraint Satisfaction Problem (CSP) encodings and techniques can be found in therein. SAT techniques have been adapted for more expressive logics. For instance, in the case of Satisfiability Modulo Theories (SMT), the problem is to decide the satisfiability of a formula with respect to a decidable background theory, such as the theory of linear (integer or real) arithmetic, arrays, lists, etc., or combinations of them, in first order logic with equality [14]. Input formulas are often syntactically restricted, for example, to be quantifier-free, so that the problem is still decidable. Hence, an SMT instance is a generalization of a Boolean SAT instance in which some propositional variables have been re- placed by predicates from the underlying theories, and can contain formulas Adaptations of SAT techniques to the SMT framework have been described therein. The main application area of SMT is hardware and software verification. However, the available theories do not restrict the usage of SMT to verification problems and, in fact, they allow to encode many problems outside the verification area in a very natural way. There are already promising results in the direction of adapting SMT techniques for solving CSPs, even in the case of combinatorial optimization (see, e.g., [10] for an application of an SMT solver on an optimization problem, being competitive with the best weighted CSP solver with its best heuristic on

that problem). Fundamental challenges on SMT for Constraint Programming (CP) and Optimization are detailed in [11]. Since the beginning of CSP solving, its holy grail has been to obtain a declarative language that allows users to easily specify their problem and forget about the techniques required to solve it.

3 Problem statement

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some background first-order theory. That is, an SMT instance is a first-order formula where some function and predicate symbols have interpretations, according to the background theories. Examples of theories are Equality and Uninterrupted Functions, Linear Integer Arithmetic, Linear Real Arithmetic, their fragments Integer Difference Logic and Real Difference Logic, Arrays (useful in modeling and verifying software programs), Bit-Vectors (useful in modeling and verifying hardware designs), or combinations of them (see [13] for details). Most SMT solvers are restricted to decidable quantifier free fragments of their logics, but this success for many applications. There are two main approaches to solve SMT instances, namely, the eager and the lazy approach. In the eager approach, the formula is translated into an propositional formula. This allows the use of o_$the-shelf SAT solvers, but has important drawbacks like, e.g., exponential memory blow-ups. For this reason, mo$
$of-the-art SMT solver simple mental azy approach, which does not involve a translation into SAT. One of thes$
$For dalgorithm. For this reason, many solvers give a special treatment to such kind of literals.$

4 The Scope

The input language deals with formulas, global constraints and the If Then Else constraint. The first part of a comprehension list is the pattern, i.e., the expression that we want to generate. Currently, patterns must be arithmetic expressions (in this example, the elements of the bidimensional array m). The rest of the comprehension list is formed by two distinct kinds of expressions, namely, the generators (in the example, i in [1..3] and j in [1..3], that expand the pattern) followed by the filters, that restrict these expansions (e.g., i¡¿j).

5 Methodology: Compilation

The compiler has been implemented in Haskell. The compilation process has two steps: the first step only checks for syntactic compliance and some minor semantic details, and generates an intermediate code. The second step is the one in charge of semantic analysis and the final SMT-LIB code generation. This code generation step distinguishes between expressions that must be evaluated at compilation time (such as, for instance, the expressions in the condition of the If-Then-Else statement), or translated into SMT-LIB expressions (for instance a basic constraint). The names of the variables are preserved from the input.

6 Conclusion

We have presented Simply, a tool for easy CSP modeling and solving, whose main novelty is the generation of SMT problem instances in the standard SMT- LIB format as output. Our aim is to take advantage from the improvements that take place from year to year in SMT technology and methods, in order to solve CSPs. Our tool can also serve as a CSP benchmark generator for SMT solvers comparison. However, much work

is still to be done in the development of Simply to make it competitive with other tools for CSP solving. We distinguish among three aspects.

Your text goes here.

1.1 A subsection

More text. 7. References

Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: an executable specification language for solving all problems in NP. Computer Languages, 26(24):165195, July 2000.