tascpl: TAS Solver for Classical Propositional Logic

M. Ojeda-Aciego and A. Valverde

Dept. Matemática Aplicada. Universidad de Málaga.
{aciego,a_valverde}@ctima.uma.es

Abstract. We briefly overview the most recent improvements we have incorporated to the existent implementations of the TAS methodology, the simplified Δ-tree representation of formulas in negation normal form. This new representation allows for a better description of the reduction strategies, in that considers only those occurrences of literals which are relevant for the satisfiability of the input formula. These reduction strategies are aimed at decreasing the number of required branchings and, therefore, control the size of the search space for the SAT problem.

1 Overview of TAS

TAS denotes a family of refutational satisfiability testers for both classical and non-classical logics which, like tableaux methods, also builds models for non valid formulas. So far, we have described algorithms for classical propositional logic [6,1], finite-valued propositional logics [3] and temporal logics [2].

The basis of the methodology is the alternative application of reduction strategies over formulas and a branching rule; the included reduction strategies are based on equivalence or equisatisfiability transformations whose complexity is at most quadratic; when no more simplifications can be applied, then the branching strategy is used and then the simplifications are called for. The power of the method is based not only on the intrinsically parallel design of the involved transformations, but also on the fact that these transformations are not just applied one after the other, but guided by some syntax directed criteria.

1.1 Δ-Trees

The improved version of the TAS satisfiability tester for classical propositional logic, `tascpl`, presented here uses and alternative representation of the boolean formulas: the simplified Δ-tree representation [6]: in the same way that conjunctive normal forms are usually interpreted as lists of clauses, and disjunctive normal forms are interpreted as lists of cubes, we interpret negative normal forms as *trees* of clauses and cubes. In Fig. 1 an example is given in which a negation normal formula together with its Δ-tree representation are shown:

The part of the algorithm which obtains benefit from this more compact representation of negation normal formulas is the module of reduction strategies.

J.J. Alferes and J. Leite (Eds.): JELIA 2004, LNAI 3229, pp. 738–741, 2004.
© Springer-Verlag Berlin Heidelberg 2004

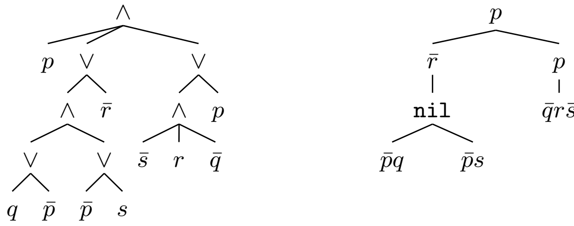

Fig. 1.

1.2 Reductions

Two types of reduction transformations to decrease the size of a negation normal form A at (at most) quadratic cost can be applied by a TAS system: meaning- and satisfiability-preserving transformations.

Restricted form. This is a generalization of the restricted form for cnfs; those subtrees in a Δ-tree which can be detected to be either valid, or unsatisfiable, or equivalent to literals are simplified.

Complete reduction. Literals in the root of a Δ-tree can be assumed true, and an equisatisfiable formula is obtained.

Pure literals deletion. Literals that always appear either positively or negatively are made true, obtaining again an equisatisfiable formula.

Subreduction. With this meaning-preserving transformation, we obtain Δ-trees such that in every branch of it there is at most one ocurrence of every atom.

1.3 Branching Process

If no simplification can be applied, then the satisfiability checking process is splitted using the following Davis-Putnam branching rule: if p is an atom in T, then T is satisfiable if and only if either $T[p/\top]$ is satisfiable or $T[\overline{p}/\top]$ is satisfiable. So, the problem is divided into two independent sub-problems that can be studied in parallel.

Although some heuristics to select the literal in the branching process have been investigated, [4], there is no yet conclusive difference in the use of either of them and thus none is applied actually in the system and the first atom of the formula is used.

1.4 Construction of Models

As tableaux systems, TAS algorithms not only check for the satisfiability, but also, if the input formula is satisfiable, a model is supplied. Models are constructed by using the deleted literals in the complete reduction, pure literal deletion and in the branching process.

2 The System for Classical Logic

The system `tascpl` has been implemented using Objective CAML version 3.06. Although the main work has been done under Mac OS X, the system can be straightforwardly ported to any Unix-like platform.

A functional language has been chosen because this is the more natural way to write the operations involved in the algorithm. On the other hand, Caml programs can easily be interfaced with other languages, in particular with other C programs or libraries; two compilation modes are supported, compilation to byte-code (for portability) and to native assembly code (for performance). The native code compiler generates very efficient code, complemented by a fast, unobtrusive incremental garbage collector. Finally, as we have mentioned above, the programs can be compiled on most Unix platforms (Mac OS X, Linux, Digital Unix, Solaris, IRIX) and well as under Windows.

2.1 The Main Program: `tascpl`

As stated previously, TAS methods are satisfiability testers (this allows also to check for validity by refutation). As a result, the required input to execute `tascpl` is a text file with the formula we want check if it is valid or satisfiable.

The command line to call the program admits a flag to turn on or off the negation of the input formula:

- `tascpl -sat` <*textfile*> checks the satisfiablity of the formula described in <*textfile*>; the possible outputs are "`Unsatisfiable`" or "`Model:` <*list of literals*>", where <*list of literals*> describes a model of the input formula (if satisfiable).
- `tascpl -val` <*textfile*> checks the validity of the formula in <*textfile*>; the possible output are: "`Valid`" or "`Countermodel:` <*list of literals*>", where <*list of literals*> describes a countermodel for the input formula, provided it is not valid.

This is an example of the content of a valid input file:

```
((s1 <-> (-a | d)) &
(o010 <-> (b & s1)) &
(s2 <-> (-c | d)) &
(o020 <-> (b & s2)))
->
((o010 <-> o020) | ((a & b & -c) | (-a & b & c & -d)))
```

Any string non starting by numbers can be used as atoms, and the following Ascii symbols are used to represent connectives: the conjunction, `&`, and the disjunction, `|`, can be used with any arity; `-` is used for negation, `->` is used for implication, and `<->` is used for biimplication. Every subformula must be enclosed by parentheses, except the negation of literals.

2.2 The dtree Utility

We have included this small utility to help understanding the Δ-tree representation.

- `dtree` <*textfile*> returns a TEX file containing the Δ-tree representation of the formula in <*textfile*>; if this file is compiled with LATEX, a pretty-printed version of the tree is obtained. Obviously, this utility is only interesting for small formulas, even although it supports larger inputs.

3 Some Comments on Performance

It is worth to say that the use of this more compact representation of formulas not only has resulted in a simpler and more straightforward implementation of the method, but also in a better performance when applied to formulas taken from the libraries of satisfiability problems.

When comparing to other propositional satisfiability testers, the first problem we faced is that only the system `HeerHugo` [5] is genuinely non-clausal (see `http://www.satlive.org`), the comparison with other provers on families of non-clausal formulas has been done indirectly through a preprocessing step in order to obtain the clause form of the problems.

TAS generally outperforms for families of formulas which are not directly stated in clause form (for instance, formulas containing a number of connectives of bi-implication) whereas the performance it not as good when applied to formulas already in clause form. This is natural, for TAS methods were designed primarily as non-clausal theorem provers.

References

1. G. Aguilera, I. P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. Reductions for non-clausal theorem proving. *Theoretical Computer Science*, 266(1/2):81–112, 2001.
2. I. P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. Implicates and reduction techniques for temporal logics. *Annals of Mathematics and Artificial Intelligence*, 27:2–23, 1999.
3. I.P. de Guzmán, M. Ojeda-Aciego, and A. Valverde. Restricted Δ-trees and reduction theorems in multiple-valued logics. *Lecture Notes in Artificial Intelligence* 2527:161–171, 2002.
4. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. *Lecture Notes in Computer Science* 2083:341–363, 2001.
5. J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000: Highlights of Satisfiability Research in the year 2000*, Frontiers in Artificial Intelligence and Applications, pages 261–281. Kluwer Academic, 2000.
6. G. Gutiérrez, I.P. de Guzmán, J. Martínez, M. Ojeda-Aciego, and A. Valverde. Satisfiability testing for Boolean formulas using Δ-trees. *Studia Logica*, 72:33–60, 2002.