*SAT System Description

Armando Tacchella
DIST, V.le Causa 13, 16145 Genova, Italy

Abstract

*SAT is a platform for the development of decision procedures for modal and description logics. Currently, *SAT handles the expressive power of $\mathcal{ALC}$ and features several decision procedures for classical modal logics. *SAT provides an open, easy to maintain, yet efficient implementation framework. These goals are achieved through a modular design and the extensive reuse of software components from state-of-the-art systems for propositional satisfiability and model checking.

1 Introduction

In this paper we present *SAT, a platform for the development of SAT-based decision procedures. By SAT-based we mean built on top of a SAT solver in the spirit of [1]. *SAT is designed to investigate the applicability of the SAT-based approach to a large class of modal and description logics. Currently, the most expressive logic that *SAT can deal with is $\mathcal{ALC}$. We have already used *SAT to experiment with the SAT-based approach in several classical modal logics ([2]). Noticeably, for some of these logics, neither tableau decision procedures nor translations to first order logic are available.

*SAT is designed from scratch both to be modular and to allow for an easy integration of new decision procedures and optimizations. *SAT is implemented in C and extensively reuses software components from state-of-the-art systems for propositional satisfiability and model checking. The *SAT propositional engine is an embedded version of SATO, one of the fastest SAT checkers currently available ([3]). We chose SATO because it features many optimizations that we exploited in *SAT and because it is well engineered, making the embedding work relatively easy. Most of *SAT data structures are built using the GLU library of data types. GLU is a part of the VIS model checking system ([4]) and it provides efficient implementations of some general purpose data types such as lists, hash-tables, sparse matrices and graphs. Taking SATO and GLU off-the-shelf, we inherit and exploit in *SAT several years of experience in building highly optimized data structures and algorithms for propositional satisfiability and model checking.

	Syntax	Semantics
Concepts	A	$\mathsf{A}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
	$\top$	$\Delta^{\mathcal{I}}$
	$\bot$	$\emptyset$
	$\neg C$	$\Delta^{\mathcal{I}} - C^{\mathcal{I}}$
	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	$\exists R.C$	$\{d \in \Delta^{\mathcal{I}} : R^{\mathcal{I}}(d) \cap C^{\mathcal{I}} \neq \emptyset\}$
	$\forall R.C$	$\{d \in \Delta^{\mathcal{I}} : R^{\mathcal{I}}(d) \subseteq C^{\mathcal{I}}\}$

Figure 1: Description Logic for *SAT

2 Language

*SAT implements the description logic shown in Figure 1. In the syntax chart A is a concept name; C and D are arbitrary concepts; R is an arbitrary role. The semantics for *SAT is a normal semantics for description logics, with a domain $\Delta^{\mathcal{I}}$, and an interpretation function $^{\mathcal{I}}$ for concepts and roles. *SAT tests satisfiability of concepts. Subsumption tests can be performed by translating them into equivalent tests for satisfiability.

*SAT handles formulae built according to the rules outlined in Figure 2. The *SAT input syntax closely resembles the one described in the KRSS document [5], but it is limited to the subset needed to reason with $\mathcal{ALC}$ descriptions. *SAT does not allow concepts and role names to be introduced, nor does it have facilities to handle complex data bases representation and querying.

3 Structure and implementation

*SAT modular architecture is depicted in Figure 3. The thickest box represents the whole system and, inside it, each box represents a different module. Solid boxes represent actual modules, while dashed boxes stand for homogeneous groups of routines inside a module. Dotted

```
formula::=boolean_expression |
          rule_expression    |
          atomic_expression;
boolean_expression::=(unary_op formula) |
                     (binary_op formula formula) |
                     (nary_op formula formula_list);
formula_list::=formula form_list | ϵ;
rule_expression::=(rule_op rule formula);
atomic_expression ::= atom | TOP | BOTTOM;
unary_op ::= NOT;
binary_op ::= IFF | IMP;
nary_op ::= AND | OR;
rule_op ::= ALL | SOME;
atom ::= C0 | ... | Cn;
rule ::= R0 | ... | Rn;
```

Figure 2: *SAT input grammar

boxes are placeholders for future extensions. Notice that the module DPSAT glues together the main data structure (DAG), the propositional engine (SAT SOLVER), and the logic dependent modules ($\mathcal{ALC}$ and possibly other ones). DPSAT provides software "plugs" where new logic dependent modules and/or different SAT modules can be inserted independently of each other. The dashed horizontal lines single out the four main parts of *SAT:

INTERFACE: The modules KRIS, KSATC and LWB are parsers for different input syntaxes. The module TREE implements formula trees and some simple preprocessing routines.

DATA: The module DAG (for *Directed Acyclic Graph*) implements the main data structure of *SAT. Storage and internal transformation of the input formula are supported by DAG routines.

ENGINE: This part includes the module SAT SOLVER, the propositional core of *SAT. Currently, this module is an embedded version of SATO 3.2 (see [3]). The dashed box (labeled CNF) stands for a set of DPSAT routines that implement a propositional CNF converter.

LOGICS: Currently, *SAT features a module for $\mathcal{ALC}$ logic and some modules for classical modal logics that are omitted in Figure 3. The dotted boxes identified with "*" will be instantiated to expressive description logics.

To understand the behavior of *SAT, let φ be the formula $\forall R.(\forall R.C_1 \wedge \forall R.C_2) \vee \neg\forall R.C_2$. *SAT first stores φ into an intermediate representation (provided by TREE) where it undergoes some preliminary transformations (e.g. $\exists R.\psi$ becomes $\neg\forall R.\neg\psi$). Then, building of the internal representation (provided by DAG) causes lexical normalization and propositional simplification to be performed on φ. The resulting data structure is depicted in Figure 4 (left). Next, *SAT creates a look up table (LUT) that establishes a correspondence between subformulae of the form $\forall R.\psi$ and newly added atoms C_ψ. The

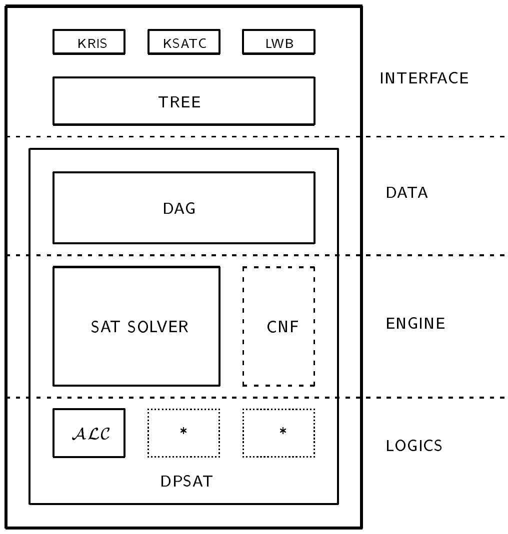

Figure 3: *SAT modular architecture

result is depicted in Figure 4 (right), where the numbers appearing in the LUT have the obvious meaning. Notice that the top-level formula $\varphi_0 = C_5 \vee \neg C_3$ is now purely propositional. If SAT SOLVER accepts only CNF formulae then (i) for every LUT entry C_ψ, both ψ and $\neg\psi$ are converted to CNF and (ii) the top level formula φ_0 is replaced by its CNF conversion. Finally, the core decision process starts. SAT SOLVER is properly initialized and called with φ_0 as input. Once a satisfying truth assignment is found, a logic dependent module (e.g. $\mathcal{ALC}$) is called to check for its consistency. The recursive tests are built in constant time using the LUT to reference the subformulae. The process continues until no more truth assignments are possible or a model is found ([1] details this process for $\mathcal{ALC}$).

4 Features

It is common for DL reasoners to incorporate several optimizations to speed up the search. State of the art systems, such as DLP ([6]) and FaCT ([7]), introduce techniques like semantic branching, boolean constraint propagation (BCP) and heuristic guided search as optimizations on top of a tableau-based algorithm. In *SAT, the module SAT SOLVER implements a Davis Putnam algorithm and such techniques are inherited for free. Moreover, *SAT features:

Preprocessing: the module TREE includes simple preprocessing routines, e.g., flattening of nested binary and/or trees to and/or lists. Internal representation as a DAG prevents from duplicating the storage for common subformulae and from assigning different atoms to the same subformula. Also, DAG constructors eliminate analytically unsatisfiable subformulae.

CNF conversion: CNF routines allow *SAT to handle any formula even if SAT SOLVER accepts CNF formulae only. The explosion of storage requirements is avoided through the use of *renaming* (see, e.g., [8]), i.e., the conversion adds new variables that stand for entire subformulae. Notice that adding variables has a limited impact since (i) it is impossible to generate different variables for the same subformula, and (ii) we can avoid the search on added variables by suitably modifying the splitting heuristics.

Splitting heuristics: SATO comes with six different splitting heuristics. We inherited all of them in our setting and we made them available for experimentation. All of the heuristics are modified to cope with the variables added by the CNF conversion. Currently, they give low ranks to the added variables to force branching on the original variables first. This prevents search on the added variables as long as they are detected and simplified by BCP.

Modal pruning: delaying the consistency check of the assignments built by the SAT solver until the propositional search is done may cause *trashing*, i.e., the solver repeatedly generates different assignments that contain the same inconsistent kernel. To prevent this, two different strategies can be devised. The first one, called *early pruning* in [1], checks incomplete assignments before every propositional branch and it enforces backtracking on inconsistencies. The second one, that we called *modal backjumping*, waits for inconsistencies to arise in the final consistency check and then looks for the smallest inconsistent kernel, possibly using different search strategies. The search is then forced back to the branch that generated the inconsistency. *SAT features both strategies, since both are effective in pruning the search space, although early pruning performs better on constrained, i.e., likely unsatisfiable, formulae, while modal backjumping performs better on unconstrained, i.e., likely satisfiable, formulae.

5 Future work

Investigation of the applicability of the SAT-based approach to description logics is an on-going project. As such, *SAT is under continuing development. It currently handles $\mathcal{ALC}$ description logic and we plan to extend it to more expressive logics in the spirit of [7] and [6]. We also plan to incorporate new optimizations such as different kinds of caching and domain-specific splitting heuristics. *SAT is available via the WWW under:

`http://www.mrg.dist.unige.it/~tac/StarSAT.html`

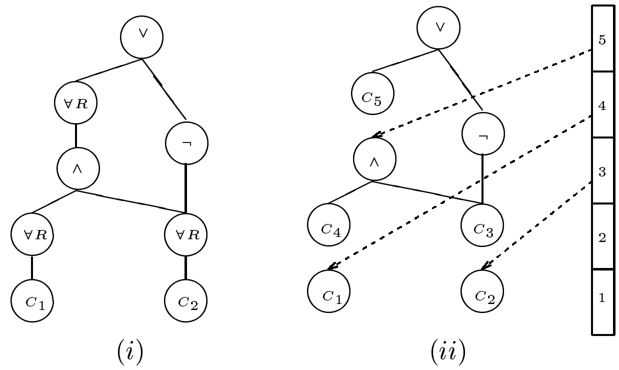

Figure 4: Internal representation of concepts in *SAT

Acknowledgements

I am grateful to Hantao Zhang and the VIS group for making their source code available. I wish to thank Enrico Giunchiglia and the other members of the *SAT project for giving me valuable suggestions and Jeff B. Golden for checking the final version of this paper.

References

[1] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning - KR'96*, Cambridge, MA, USA, November 1996. Also DIST-Technical Report 9607-08 and IRST-Technical Report 9601-02.

[2] E. Giunchiglia, F. Giunchiglia, and A. Tacchella. *SAT, KSATC, DLP and TA: a comparative analysis. In *Collected Papers from the International Description Logics Workshop (DL'99)*. CEUR, July 1999.

[3] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July13–17 1997. Springer.

[4] VIS-group. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification - CAV-96*, pages 428–432, 1996.

[5] Peter Patel-Schneider and Bill Swartout. Description-logic knowledge representation system specification. Technical report, AT&T Bell Laboratories, 1993.

[6] P. F. Patel-Schneider. DLP system description. In E. Franconi, G. De Giacomo, R. M. MacGregor, W. Nutt, C. A. Welty, and F. Sebastiani, editors, *Collected Papers from the International Description Logics Workshop (DL'98)*, pages 87–89. CEUR, May 1998.

[7] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.

[8] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.