System Description: *SAT
A platform for the development of modal decision procedures ⋆

Enrico Giunchiglia and Armando Tacchella

DIST, Università di Genova, Viale Causa 13 – 16145 Genova, Italy
{enrico,tac}@dist.unige.it

Abstract. *SAT is a platform for the development of modal decision procedures. Currently, *SAT features decision procedures for the normal modal logic K(m) and for the classical modal logic E(m). *SAT embodies a state of the art SAT solver, and includes techniques for optimizing automated deduction in modal and temporal logics. Owing to its modular design and to the extensive reuse of software components, *SAT provides an open, easy to maintain, yet efficient implementation framework.

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]. Currently, *SAT features SAT-based decision procedures for the normal modal logic K(m), and for the classical modal logic E(m) [9, 2].

The *SAT propositional engine is an embedded version of SATO 3.2, one of the most efficient SAT checkers publicly available [3]. We chose SATO because it is a fast propositional reasoner and it features many optimizations that we exploited in *SAT. We also implemented other optimizations that speed up modal reasoning, like:

- early investigation of modal successors [1],
- internal optimized clause form conversions [2], and
- caching structures and retrieval algorithms [4].

*SAT has been designed to be modular and to allow for an easy integration of new decision procedures and optimizations. The system is implemented in C and extensively reuses software components from state-of-the-art systems, i.e., SATO, and the GLU library of data types from the VIS model checking system [5]. The GLU library provides *SAT with efficient implementations of, e.g., lists, hash-tables, sparse-matrices. 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 automated deduction in propositional and temporal logics.

*SAT source code, documentation and experimental results are available on the WWW at:

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

⋆ We wish to thank Fausto Giunchiglia, Peter Patel-Schneider and Roberto Sebastiani for useful discussions related to *SAT. This work is supported by MURST.

```
function LSAT(φ)            return LSAT_DP(cnf(φ), ⊤).

function LSAT_DP(Φ, μ)
   if Φ = ∅ then return LCONSIST(μ);                  /* base */
   if ∅ ∈ Φ then return False;                        /* backtrack */
   if { a unit clause {l} is in Φ }                   /* unit */
      then return LSAT_DP(assign(l, Φ),μ ∧ l);
   if not LCONSIST(μ) then return False;              /* early pruning */
   l := choose-literal(Φ, μ);
   return LSAT_DP(assign(l, Φ),μ ∧ l) or              /* split */
          LSAT_DP(assign(l̄, Φ),μ ∧ l̄).
```

Fig. 1. LSAT and LSAT_{DP}

2 Algorithms

For sake of clarity, we introduce some preliminary notions. The set of formulas is constructed starting from a given set of propositional letters and applying the 0-ary operators $\top$ and $\bot$ (representing truth and falsity respectively); the unary operators $\neg$ and $\Box$; and the binary operators $\wedge$, $\vee$, $\supset$ and $\equiv$.[1] A *modal logic* is a set of formulas (called *theorems*) closed under tautological consequence. A formula φ is *consistent* in a modal logic L (or L-*consistent*) if $\neg\varphi$ is not a theorem of L, i.e., if $\neg\varphi \notin$ L. By *atom* we mean a propositional letter or a formula of the form $\Box\varphi$. A *literal* is either an atom or the negation of an atom. An *assignment* is any conjunction μ of literals such that for any pair ψ, ψ' of conjuncts in μ, it is not the case that $\psi = \neg\psi'$. An assignment μ *satisfies* a formula φ if μ entails φ by propositional reasoning.

Consider a formula φ. Let L be a modal logic. Whether φ is L-consistent can be determined by implementing two mutually recursive procedures:

– LSAT(φ) for the generation of assignments satisfying φ, and
– LCONSIST(μ) for testing the L-consistency of each generated assignment μ.

The procedure LSAT is independent of the particular modal logic L considered, and can be based on any propositional decision procedure (see [2]). Indeed, the logic specific reasoning is delegated to LCONSIST. Currently, *SAT features the procedures ECONSIST and KCONSIST playing the role of LCONSIST for the logics E(m) and K(m) respectively. For lack of space, we present only the LSAT algorithm here. For ECONSIST and KCONSIST, see [2] and also [4].

The LSAT procedure implemented in *SAT is based on SATO 3.2 [3], an efficient implementation of the Davis-Putnam-Longemann-Loveland (DP) procedure [6]. Figure 1 shows an high-level description of LSAT. In the Figure:

– *cnf*(φ) is a set of clauses obtained from φ by applying a conversion to conjunctive normal form (CNF) based on renaming (see, e.g., [7]).

[1] For simplicity, we consider the case with only one modality.

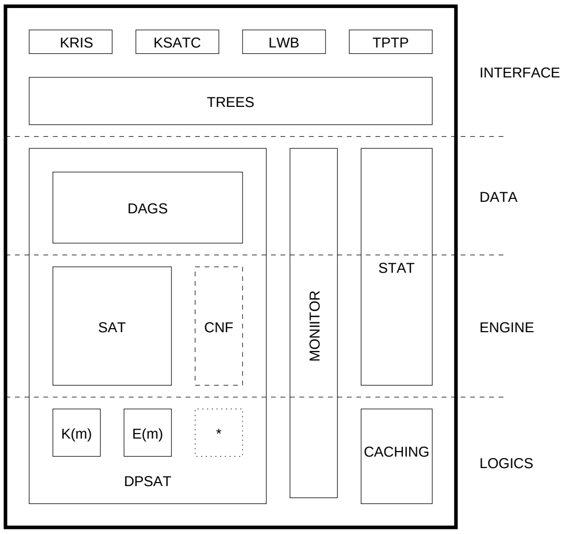

Fig. 2. *SAT modular architecture

- *choose-literal*(Φ, μ) returns a literal occurring in Φ and chosen according to some heuristic criterion.
- if l is a literal, $\bar{l}$ stands for A if $l = \neg A$, and for A if $l = \neg A$;
- for any literal l and set Φ of clauses, $assign(l, \Phi)$ is the set of clauses obtained from Φ by (i) deleting the clauses in which l occurs as a disjunct, and (ii) eliminating $\bar{l}$ from the others.

As can be observed, the procedure LSAT_{DP} in Figure 1 is the DP-procedure modulo (i) the call to LCONSIST(μ) when it finds an assignment μ satisfying the input formula ($\Phi = \emptyset$), and (ii) the *early pruning* step, i.e., a call to LCONSIST(μ) that forces backtracking after each unit propagation when incomplete assignments are not L-consistent. Early pruning prevents *SAT from *thrashing*, i.e., from repeatedly generating different assignments that contain a same inconsistent kernel [1, 8].

3 Implementation and features

The *SAT modular architecture is depicted in Figure 2. The thickest external box represents the whole system and, inside it, each solid box represents a different module. By module, we mean a set of routines dedicated to a specific task.[2] The dashed horizontal lines single out the four main parts of *SAT:

INTERFACE: The modules KRIS, KSATC, LWB, and TPTP are parsers for different input syntaxes. The module TREES stores the input formula as a tree, at

[2] As a matter of fact, each module corresponds to a file in *SAT distribution package.

the same time performing some simple preprocessing (e.g. pushing negations down to atoms).

DATA: The module DAGS (for *Directed Acyclic Graphs*) implements the main data structures of *SAT. The input formula is preprocessed and stored as a DAG. A Look Up Table (LUT), mapping each atom $\Box\psi$ into a newly introduced propositional letter C_ψ is built. Then, each modal atom is replaced by the corresponding propositional letter. The initial preprocessing allows to map trivially equivalent[3] modal atoms into a single propositional letter, thus fostering the detection of (un)satisfiable subformulae [1, 10].

ENGINE: This part includes the module SAT, the propositional core of *SAT. Since SAT implements a DP algorithm, techniques like semantic branching, boolean constraint propagation (BCP) and heuristic guided search are inherited for free by *SAT. The dashed box (labeled CNF) stands for a set of DPSAT routines implementing CNF conversions based on renaming. CNF routines allow *SAT to handle any formula even if the SAT decider accepts CNF formulae only.

LOGICS: Currently, *SAT features the two modules K(m) and E(m), implementing KCONSIST and ECONSIST respectively. The dotted box "*" is a placeholder for other L-consistency modules that will be implemented in the near future. CACHING implements data structures and retrieval algorithms that are used to optimize the L-consistency checking routines contained in the logic dependent modules (see [2] for caching in E(m) and [4] for caching in K(m)).

The modules DPSAT, MONITOR and STAT span across different parts of *SAT. DPSAT interfaces the inner modules between them. The result is that these modules are loosely coupled and can be modified/replaced (almost) independently from each other. MONITOR records information about *SAT performance, e.g, cpu time, memory consumption, number of L-consistency checks. STAT explores the preprocessed input formula and provides information like number of occurrences of a variable, number of nested boxes. This information is used by different modules, e.g., for dimensioning the internal data structures.

To understand the behavior of *SAT, let φ be the formula $\neg(\neg\Box(\Box C_2 \wedge \Box C_1) \wedge \Box C_2)$. *SAT first stores φ as an intermediate representation (provided by TREES) where it undergoes some preliminary transformations. In our case, φ becomes $(\Box(\Box C_2 \wedge \Box C_1) \vee \neg\Box C_2)$. Then, the building of the internal representation (provided by DAGS) causes lexical normalization ($(\Box C_2 \wedge \Box C_1)$ would be rewritten into $(\Box C_1 \wedge \Box C_2)$) and propositional simplification (e.g., $(C \vee C)$ would be rewritten into C) to be performed on φ. The resulting formula is represented by the data structure depicted in Figure 3 (left). Next, *SAT creates the LUT and replaces each modal atom with the corresponding propositional letter. The result is depicted in Figure 3 (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

[3] Technically, the preprocessing maps a formula φ into a formula φ' which is logically equivalent to φ in any classical modal logic (see [9] for the definition of classical modal logic).

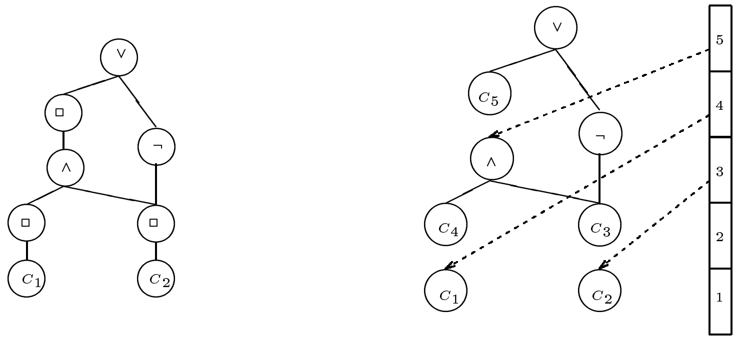

Fig. 3. Internal representation of concepts in *SAT

now purely propositional. If SAT 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 is properly initialized and called with φ_0 as input. Once a satisfying truth assignment is found, a logic dependent module (e.g. K(m)) is called to check its L-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 ([2] details this process for K(m), E(m) and several other classical modal logics).

References

1. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE-96*, LNAI, 1996.
2. E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-Based Decision Procedures for Classical Modal Logics. To appear in *Journal of Automated Reasoning*.
3. H. Zhang. SATO: An efficient propositional prover. In *Proc. CADE-97*, volume 1249 of *LNAI*, 1997.
4. E. Giunchiglia and A. Tacchella. Subset-matching Size-bounded Caching for Satisfiability in Modal Logics, 2000. Submitted.
5. VIS-group. VIS: A system for verification and synthesis. In *Proc. CAV-96*, LNAI, 1996.
6. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
7. D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
8. I. Horrocks. Optimizing Tableaux Decision Procedures for Description Logics. PhD thesis, University of Manchester, 1997.
9. B. F. Chellas. *Modal Logic – an Introduction*. Cambridge University Press, 1980.
10. U. Hustadt and R.A. Schmidt. On evaluating decision procedures for modal logic. In *Proc. IJCAI-97*, 1997.