Formal verification of a generic framework to synthesize SAT-provers

Francisco–Jesús Martín–Mateos, José–Antonio Alonso, María–José Hidalgo and José–Luis Ruiz–Reina

Computational Logic Group
Dept. of Computer Science and Artificial Intelligence, University of Seville
E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
E-mails: {fmartin,jalonso,mjoseh,jruiz}@cs.us.es

Abstract. We present in this paper an application of the ACL2 system to generate and reason about propositional satisfiability provers. For that purpose, we develop a framework where we define a generic SAT-prover based on transformation rules, and we formalize this generic framework in the ACL2 logic, carrying out a formal proof of its termination, soundness and completeness. This generic framework can be instantiated to obtain a number of verified and executable SAT-provers in ACL2, and this can be done in an automated way. Three instantiations of the generic framework are considered: semantic tableaux, sequent and Davis–Putnam–Logeman–Loveland methods.

1. Introduction

A common practice in program verification is stepwise refinement. This means that essential properties of programs can be first proved at a very abstract level, considering only a generic specification of the program, skipping technical details of concrete implementations. Thus, the properties proved can be deduced for a given implementation of the generic specification, by simply showing that this implementation is a concrete instance of the generic procedure. Further refinements of the implementations (in order to obtain better performance) can still be verified by showing that they compute the same results as an-other verified implementation. In this paper, we describe an application of this technique to reason formally about a family of propositional satisfiability (SAT) decision procedures, using the ACL2 system.

SAT provers are an important component of many applications in theorem proving in particular and artificial intelligence in general [9], so it makes sense the development of formally verified SAT decision procedures, as a way of certifying this "proof engine" component [18].

The reason why we have chosen ACL2 as the logic and prover used to reason about this procedures, is that this system provides a framework

* This work has been supported by project TIC2000-1368-C03-02 (Ministry of Science and Technology, Spain), cofinancied by FEDER founds.

where reasoning and computing can be done. ACL2 [12] is a programming language, a logic for reasoning about programs in the language, and a theorem prover supporting formal reasoning in the logic. So the procedures can be implemented, executed and formally verified in the same system.

Three case studies are considered: semantic tableaux, sequent calculus and the Davis–Putnam–Logeman–Loveland method. The common pattern of all these SAT procedures is that they can be described as rule based transformation systems. For that purpose, we develop a generic framework into which these SAT-provers can be placed. A generic SAT-prover is formalized in ACL2 and its main properties are proved; using functional instantiation, concrete instances of the generic framework can be defined to obtain formally verified and Common Lisp executable SAT-provers. As a byproduct, we have developed a tool to make the instantiation process more convenient, obtaining *in an automated way* the concrete and executable procedures and the instances of the theorems proved for the generic framework.

This paper is an extended and revised version of [15]. It is organized as follows. In Section 2 we define a generic framework in order to build a generic transformation based SAT-prover, and we sketch a proof of its termination, soundness and completeness properties. We also describe how three well-known SAT-provers methods (tableaux, sequent calculus and Davis–Putnam–Logeman–Loveland method) can be placed into the generic framework. In Section 3 we show how this framework has been formalized in ACL2 and how its main properties has been proved. In Section 4 we describe how these generic definitions and theorems has been instantiated, to obtain verified and executable Common Lisp definitions of tableaux based, sequent based and Davis–Putnam–Logeman–Loveland SAT-provers. Finally, in Section 5 we draw some conclusions.

Due to the lack of space we will skip details of the mechanical proofs and for the same reason some function definitions will be omitted. The complete formalization is available in [16].

2. A generic framework to develop propositional SAT-provers

Analyzing some well-known methods of proving propositional satisfiability (such as sequent, tableaux or Davis–Putnam–Logeman–Loveland), we can observe a common behavior. They do not work directly on formulas but on objects built from formulas. The objects are repeatedly modified using expansion rules, reducing their complexity

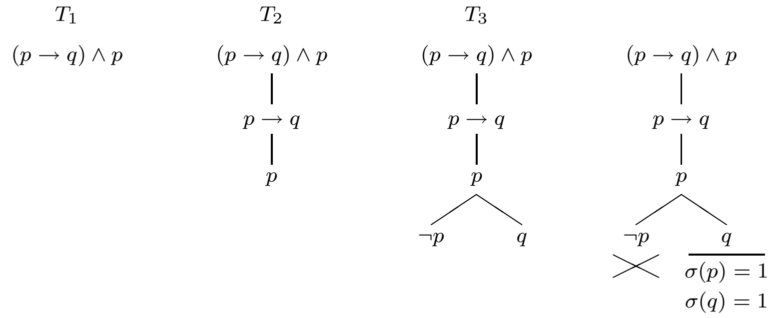

Figure 1. An example of tableaux method

in such a way that their meaning is preserved. Eventually, from some kind of simple objects, a *distinguished valuation* proving satisfiability of the original formula can be obtained. If no such object is found, then unsatisfiability of the original formula is proved. We must point out that these objects are not only theoretical structures used to describe the method (e.g. lists or sets of formulas), but they can be real data structures used in the implementation (e.g. arrays, linked lists, hash tables, ...) of the SAT procedures.

We can see this behavior in the semantic tableaux method, by means of the example shown in Figure 1. From the formula $(p \to q) \wedge p$ the initial tree T_1 with a single node is built. In a first step the formula is expanded obtaining one extension with two formulas $p \to q$ and p (tree T_2). In a second step the formula $p \to q$ is expanded obtaining two extensions, the first with the formula $\neg p$ and the second with the formula q (tree T_3). The left branch becomes closed (*i.e.*, with complementary literals) and the right one provides a model σ. Thus, the tableaux method can be seen as the application of a set of expansion rules acting on branches of trees (the objects) until a branch without complementary literals is obtained. For this branch, a distinguished valuation (making that branch true) is easily obtained. Otherwise, all branches are closed and unsatisfiability is proved.

Our goal in this section is to describe a generic framework where these methods can be fit. First we introduce some notation. We consider an infinite set of proposition symbols Σ and a set of truth values, $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$, where $\mathbf{t}$ denotes *true* and $\mathbf{f}$ denotes *false*. $\mathbb{P}(\Sigma)$ denotes the set of *propositional formulas* on Σ (the truth values are not considered as formulas), where the basic connectives are $\neg$, $\wedge$, $\vee$, $\to$ and $\leftrightarrow$. The *complement* of a formula F, denoted as $\overline{F}$, is defined such that $\overline{F} = G$ if $F = \neg G$, and $\overline{F} = \neg F$ otherwise. A *literal* is a formula p or $\neg p$, where $p \in \Sigma$. A *clause* is a finite sequence of literals. A *valuation* is a function

$\sigma : \Sigma \longrightarrow \mathbb{B}$; we denote $\mathbb{V}_\Sigma$ the set of all valuations defined on Σ. The valuations are extended to $\mathbb{P}(\Sigma)$ in the usual way. We denote $\sigma \models F$ when $\sigma(F) = \mathbf{t}$, and we say that σ is a *model* of F. A valuation σ is a model of a clause C, if it is a model of some literal in C. The capital Greek letters Γ and Δ (possibly with subscripts) denote finite sequences of formulas (we sometimes use the term *list* instead of finite sequence). We will use the notation $\langle e_1, ..., e_k \rangle$ to represent a finite sequence, and O^* to denote the set of finite sequences of elements of the set O. We say that x is a member of the list $\langle e_1, ..., e_k \rangle$, denoted as $x \in \langle e_1, ..., e_k \rangle$, if $\exists i, 1 \leq i \leq n$, such that $x = e_i$. We write $\langle \Gamma_1, F, \Gamma_2 \rangle$ or Γ_1, F, Γ_2, to distinguish the formula F in a sequence of formulas. Finally, $\mathcal{O}rd$ denotes the class of all ordinals.

2.1. A generic algorithm for proving propositional satisfiability

DEFINITION 1. *A* Propositional Transformation System *(PTS, for short) is a triple* $\mathcal{G} = \langle \mathcal{O}_\mathcal{G}, \rightsquigarrow_\mathcal{G}, \models_\mathcal{G} \rangle$, *where* $\mathcal{O}_\mathcal{G}$ *is a set, and* $\rightsquigarrow_\mathcal{G}$ *and* $\models_\mathcal{G}$ *are binary relations such that* $\rightsquigarrow_\mathcal{G} \subseteq \mathcal{O} \times (\mathcal{O}^* \cup \{\mathbf{t}\})$ *and* $\models_\mathcal{G} \subseteq \mathbb{V}_\Sigma \times \mathcal{O}$.

We will call $\mathcal{O}_\mathcal{G}$ the set of *propositional objects* (or simply *objects*) and $\rightsquigarrow_\mathcal{G}$ the set of *expansion rules*. Intuitively, the objects are the structures used by a propositional SAT-prover and the expansion rules describe the steps that it performs. Note that we allow rules of the form $O \rightsquigarrow_\mathcal{G} \langle\rangle$ and rules of the form $O \rightsquigarrow_\mathcal{G} \mathbf{t}$. The first one represents dead ends in the search for satisfiability, and the second one represents successful ends. When $\sigma \models_\mathcal{G} O$, we say that σ is a *distinguished valuation* for O. The idea is that when a successful end is found, the distinguished valuations for the last object provide a model of the original formula. Intuitively, the relation $\models_\mathcal{G}$ translates the relation $\models$ from formulas to the objects used by the SAT-prover.

DEFINITION 2. *Given a PTS* $\mathcal{G} = \langle \mathcal{O}_\mathcal{G}, \rightsquigarrow_\mathcal{G}, \models_\mathcal{G} \rangle$*:*

1. *A* computation rule *is a function* $r : \mathcal{O} \longrightarrow \mathcal{O}^* \cup \{\mathbf{t}\}$ *such that* $r \subseteq \rightsquigarrow_\mathcal{G}$.

2. *A* representation function *is a function* $i : \mathbb{P}(\Sigma) \longrightarrow \mathcal{O}$.

3. *A* measure function *is a function* $\mu : \mathcal{O} \longrightarrow \mathcal{O}rd$.

4. *A* model function *is a function* $\gamma : \mathcal{O}_\mathbf{t} \longrightarrow \mathbb{V}_\Sigma$, *where* $\mathcal{O}_\mathbf{t} = \{O \in \mathcal{O} : O \rightsquigarrow_\mathcal{G} \mathbf{t}\}$.

Given a PTS $\mathcal{G} = \langle \mathcal{O}_\mathcal{G}, \rightsquigarrow_\mathcal{G}, \models_\mathcal{G} \rangle$, a computation rule r and a representation function i, we define the following algorithm $SAT_\mathcal{G}$ for proving satisfiability of a propositional formula.

ALGORITHM 1 ($SAT_\mathcal{G}$). *The input to this algorithm is a propositional formula F and it proceeds as follows:*

1. *Let $L = \langle i(F) \rangle$.*
2. *While L is a non-empty list, do:*
   Select O_j a member of the list $L = \langle O_1, ..., O_n \rangle$.
   a) *If $r(O_j) = \mathbf{t}$, then stop and return $\langle O_j \rangle$.*
   b) *If $r(O_j) = \langle O'_1, ..., O'_m \rangle$ ($m \geq 0$),*
   then let $L = \langle O'_1, ..., O'_m, O_1, ..., O_{j-1}, O_{j+1}, ..., O_n \rangle$.
3. *Return $\mathbf{f}$.*

The intuitive idea is simple: given F, we start with the initial object $i(F)$ and repeatedly apply the expansion rules until $\mathbf{t}$ is obtained or until there are no more objects left. Termination of this process will be guaranteed by a measure function μ. The strategy to apply the rules is determined by the given computation rule r and by the selection strategy of objects of the list L. Note that assuming the existence of a computation rule means that for every object there is at least one rule that can be applied to it.

It can be proved that under some conditions that we give below, if $\mathbf{t}$ is obtained from an object O_j, then we can obtain a distinguished valuation using a model function and this valuation turns out to be a model of the original formula. Under the same conditions, if $\mathbf{f}$ is obtained, the original formula is unsatisfiable.

DEFINITION 3. *We say that $SAT_\mathcal{G}$ is* complete *if for all $F \in \mathbb{P}(\Sigma)$ such that $\exists \sigma \in \mathbb{V}_\Sigma : \sigma \models F$, then $SAT_\mathcal{G}(F) \neq \mathbf{f}$. We say that it is* sound *if for all $F \in \mathbb{P}(\Sigma)$ such that $SAT_\mathcal{G}(F) \neq \mathbf{f}$, then $\exists \sigma \in \mathbb{V}_\Sigma : \sigma \models F$.*

THEOREM 1. *Let $\mathcal{G} = \langle \mathcal{O}_\mathcal{G}, \rightsquigarrow_\mathcal{G}, \models_\mathcal{G} \rangle$ be a PTS, r a computation rule, i a representation function, μ a measure function and γ a model function, such that the following properties hold:*

$\mathcal{P}_1$: $O_i \in r(O) \Longrightarrow \mu(O_i) < \mu(O)$

$\mathcal{P}_2$: $F \in \mathbb{P}(\Sigma) \Longrightarrow (\sigma \models F \Longleftrightarrow \sigma \models_\mathcal{G} i(F))$

$\mathcal{P}_3$: $O \in \mathcal{O} \wedge r(O) \neq \mathbf{t} \Longrightarrow (\sigma \models_\mathcal{G} O \Longleftrightarrow \exists O_i \in r(O), \sigma \models_\mathcal{G} O_i)$

$\mathcal{P}_4$: $O \in \mathcal{O} \wedge r(O) = \mathbf{t} \Longrightarrow \gamma(O) \models_{\mathcal{G}} O$

then the algorithm $SAT_{\mathcal{G}}$ terminates for any formula and is complete and sound. Furthermore, if $SAT_{\mathcal{G}}(F) = \langle O \rangle$ then $\gamma(O) \models F$.

Termination Proof. In the termination proof of $SAT_{\mathcal{G}}$, we will use a multiset relation built from the measure function. Roughly speaking, a finite multiset over A is a subset of A "with repeated elements". Let us briefly recall the notion of multiset relation. Given a relation $<$ on a set A, we define the multiset relation induced by $<$ on the set of finite multisets over A, denoted as $<_{mul}$, in the following way: $N <_{mul} M$ if there exist X, Y finite multisets over A, such that $\emptyset \neq X \subseteq M$, $N = (M \setminus X) \cup Y$ and for all $y \in Y$ there exists $x \in X$ such that $y < x$. Intuitively, this means that a smaller multiset can be obtained by removing a non-empty subset of elements, and adding elements which are smaller than some element removed. In [7] it is proved that $<_{mul}$ is well-founded whenever $<$ is well-founded.

Let us now prove the termination of $SAT_{\mathcal{G}}$. For that purpose, we must prove that point 2 is a finite loop. Assume that the list of objects in point 2 is $\langle O_1, ..., O_n \rangle$, the selected element is O_j and $r(O_j) = \langle O'_1, ..., O'_m \rangle$, with $m \geq 0$.

We consider the relation $<_\mu$ in $\mathcal{O}$ defined as follows $O_1 <_\mu O_2$ if and only if $\mu(O_1) < \mu(O_2)$. Obviously, $<_\mu$ is a well founded relation on $\mathcal{O}$. Then, for every k, $O'_k <_\mu O_j$ by $\mathcal{P}_1$. Therefore, the multiset $\{O'_1, ..., O'_m, O_1, ..., O_{j-1}, O_{j+1}, ..., O_n\}$ is smaller than $\{O_1, ..., O_n\}$ with respect to the multiset extension of $<_\mu$ (which is also well-founded). This proves termination of $SAT_{\mathcal{G}}$.

Completeness Proof. First of all note that, by $\mathcal{P}_3$, if the algorithm reaches point 2-(b), σ is a distinguished valuation of some object in the list considered in point 2 if and only if it is a distinguished valuation of some object in the new list built in point 2-(b).

If $\sigma \models F$ then, by $\mathcal{P}_2$, $\sigma \models_{\mathcal{G}} i(F)$. Then, by the above observation, in every list considered in point 2 exists O such that $\sigma \models_{\mathcal{G}} O$. Therefore the list in point 2 cannot become empty and, since the algorithm terminates, in some step an object O' such that $r(O') = \mathbf{t}$ will be considered. Then $SAT_{\mathcal{G}}(F) = \langle O' \rangle \neq \mathbf{f}$.

Soundness Proof. If $SAT_{\mathcal{G}}(F) = \langle O \rangle$ then $r(O) = \mathbf{t}$ and, by $\mathcal{P}_4$, $\gamma(O) \models_{\mathcal{G}} O$. Then, by the property noted in the completeness proof, in every list considered in point 2 exists O' such that $\gamma(O) \models_{\mathcal{G}} O'$. Therefore, this holds for the initial list considered $\langle i(F) \rangle$, i.e., $\gamma(O) \models_{\mathcal{G}} i(F)$, and, by $\mathcal{P}_2$, $\gamma(O) \models F$.

2.2. Semantic Tableaux

We now show how the semantic tableaux method can be seen as a propositional transformation system, and how a simple SAT-prover based on this method can be seen as a particular instance of the algorithm $SAT_{\mathcal{G}}$.

Let us first overview the propositional tableaux method, following the description given in [8]. This method is a refutation system: to prove that a formula F is valid, it starts with a finite tree with only one node labeled with $\neg F$ and applies a set of expansion rules until it generates a contradiction. From a more constructive point of view, the method tries to build a model of the formula $\neg F$. If this is not possible, then F is valid.

The tableaux expansion rules are concisely presented using the uniform notation [19][1]. Using this notation, non-literal formulas are classified as doubly negated, α-formulas or β-formulas, as we show in the following tables:

Double negation	component
$\neg\neg X$	X

α	α_1	α_2
$X \wedge Y$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \rightarrow Y)$	X	$\neg Y$

β	β_1	β_2
$X \vee Y$	X	Y
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$X \rightarrow Y$	$\neg X$	Y
$X \leftrightarrow Y$	$X \wedge Y$	$\neg X \wedge \neg Y$
$\neg(X \leftrightarrow Y)$	$X \wedge \neg Y$	$\neg X \wedge Y$

Note that the α-formulas are equivalent to the conjunction of their components α_1 and α_2, the β-formulas are equivalent to the disjunction of their components β_1 and β_2, and the doubly negated formulas are equivalent to their unique component.

The method acts as follows. Let T be a finite tree, with its nodes labeled with propositional formulas, and θ a branch in T with an occurrence of a non-literal formula F. If F is $\neg\neg X$, then the branch θ is extended adding a new node labeled with X. If F is an α-formula, then the branch θ is extended adding two nodes labeled with the components α_1 and α_2 of the formula. If F is a β-formula, then the branch θ is extended producing two branches at the end, each one with a node labeled, respectively, with the components β_1 and β_2 of the formula.

A branch θ is (atomically) closed if there exist two nodes in θ labeled with complementary (literal) formulas. The method is applied until

[1] We extend the uniform notation to include equivalence.

every branch is closed. In this case the original formula F is valid. If there is a non closed branch θ such that every occurrence of a non-literal formula in θ has been expanded, then the formula $\neg F$ has a model and the formula F is not valid. In this case a model of $\neg F$ can be built from the literal formulas in θ and we say that θ provides a model. See Figure 1 for an example.

We now describe the PTS $\mathcal{T} = \langle \mathcal{O}_\mathcal{T}, \rightsquigarrow_\mathcal{T}, \models_\mathcal{T} \rangle$ associated with the semantic tableaux method. In this PTS, $\mathcal{O}_\mathcal{T}$ is the set of finite sequences of formulas (representing tableaux branches), $\sigma \models_\mathcal{T} \theta$ if and only if σ makes true every formula in the branch θ, and $\rightsquigarrow_\mathcal{T}$ is the relation described by the following rule schemata:

$\mathcal{RT}_1 : \langle \Gamma_1, G, \Gamma_2, \neg G, \Gamma_3 \rangle \rightsquigarrow_\mathcal{T} \langle\rangle$
$\mathcal{RT}_2 : \langle \Gamma_1, \neg G, \Gamma_2, G, \Gamma_3 \rangle \rightsquigarrow_\mathcal{T} \langle\rangle$
$\mathcal{RT}_3 : \langle \Gamma_1, \neg\neg G, \Gamma_2 \rangle \rightsquigarrow_\mathcal{T} \langle\langle \Gamma_1, G, \Gamma_2 \rangle\rangle$
$\mathcal{RT}_4 : \langle \Gamma_1, \alpha, \Gamma_2 \rangle \rightsquigarrow_\mathcal{T} \langle\langle \Gamma_1, \alpha_1, \alpha_2, \Gamma_2 \rangle\rangle$
$\mathcal{RT}_5 : \langle \Gamma_1, \beta, \Gamma_2 \rangle \rightsquigarrow_\mathcal{T} \langle\langle \Gamma_1, \beta_1, \Gamma_2 \rangle, \langle \Gamma_1, \beta_2, \Gamma_2 \rangle\rangle$
$\mathcal{RT}_6 : \Gamma \rightsquigarrow_\mathcal{T} \mathbf{t}$ if Γ does not have non-literal nor complementary formulas

The rule schemata $\mathcal{RT}_3$, $\mathcal{RT}_4$ and $\mathcal{RT}_5$ correspond with the tableaux expansion rules presented above. The rule schemata $\mathcal{RT}_1$ and $\mathcal{RT}_2$ check if a branch is closed and the rule $\mathcal{RT}_6$ checks if a branch provides a model.

Given concrete representation, computation rule, measure and model functions for this PTS, we define a propositional tableaux method, which we call $SAT_\mathcal{T}$, as a concrete version of the generic procedure $SAT_\mathcal{G}$. By Theorem 1, this procedure will be sound and complete if properties $\mathcal{P}_1$ to $\mathcal{P}_4$ are verified. We now define these four functions, proving the properties in passing.

The representation function $i_\mathcal{T}$ is defined such that for every $F \in \mathbb{P}(\Sigma)$, $i_\mathcal{T}(F) = \langle F \rangle$; that is, the only branch in the initial tree considered by the semantic tableaux method. Obviously, $\sigma \models F \iff \sigma \models_\mathcal{T} i(F)$ (property $\mathcal{P}_2$).

We can consider any computation rule, $r_\mathcal{T}$, such that, for every branch θ, $r_\mathcal{T}(\theta)$ is the result of applying one of the above rule schemata to θ, whenever such rule may be applied. Several versions of the semantic tableaux method could be represented by different computation rules. For example, if the rule schemata $\mathcal{RT}_1$ and $\mathcal{RT}_2$ have less priority than the others, then the expansion rules are applied until every branch is atomically closed. To finish the expansion process when the branches are closed, the rule schemata $\mathcal{RT}_1$ and $\mathcal{RT}_2$ should have higher priority than the others. Another point could be the preference order between the rule schemata $\mathcal{RT}_3$ and $\mathcal{RT}_4$, without bifurcation, and the rule schemata $\mathcal{RT}_5$, with makes a bifurcation. Taking into account

these ideas, we can define several computation rules and hence, several propositional theorem provers based on semantic tableaux associated with the above PTS.

In order to define the measure function, we define the uniform measure [1][2], denoted as u, as follows: $u(F) = 5 * \delta_{\leftrightarrow}(F) + 2 * (\delta_{\wedge}(F) + \delta_{\vee}(F) + \delta_{\rightarrow}(F)) + \delta_{\neg}(F)$, where $\delta_{\circ}(F)$ computes the number of occurrences of the connective $\circ$ in F. This measure has the following properties: $u(\alpha_1) + u(\alpha_2) < u(\alpha)$, $u(\beta_1) < u(\beta)$, $u(\beta_2) < u(\beta)$ and $u(X) < u(\neg\neg X)$. We define the measure function, $\mu_{\mathcal{T}}$, as the sum of the uniform measure of the formulas in a branch. By the properties of u, the expansion rules reduce the measure of a branch; therefore $\theta_i \in r_{\mathcal{T}}(\theta) \Longrightarrow \mu_{\mathcal{T}}(\theta_i) < \mu_{\mathcal{T}}(\theta)$ (property $\mathcal{P}_1$).

The uniform notation ensures that an α (β) formula is logically equivalent to the conjunction (disjunction) of its components and a doubly negated formula $\neg\neg X$ is also logically equivalent to X. Hence, if $\theta \rightsquigarrow_{\mathcal{T}} L$ with $L \neq \mathbf{t}$, it can be easily proved that $\sigma \models_{\mathcal{T}} \theta \iff \exists \theta_i \in L, \sigma \models_{\mathcal{T}} \theta_i$. According to our definition of computation rule, this trivially implies property $\mathcal{P}_3$.

Finally, we define the model function $\gamma_{\mathcal{T}}$ such that for every branch θ without non-literal nor complementary formulas, $\gamma_{\mathcal{T}}(\theta) \models p$ if and only if p is a positive literal occurring in θ. Obviously, if $r_{\mathcal{T}}(\theta) = \mathbf{t}$ then $\gamma_{\mathcal{T}}(\theta) \models_{\mathcal{T}} \theta$ (property $\mathcal{P}_4$).

Then, by Theorem 1, the algorithm $SAT_{\mathcal{T}}$ terminates for any formula and is complete and sound. The algorithm applied to the example of Figure 1 performs the following steps (represented as $\underset{SAT_{\mathcal{T}}}{\longmapsto}$):

$$\begin{array}{rll} \langle\langle(p \rightarrow q) \wedge p\rangle\rangle & \underset{SAT_{\mathcal{T}}}{\longmapsto} \langle\langle p \rightarrow q, p\rangle\rangle & \mathcal{RT}_3 \\ & \underset{SAT_{\mathcal{T}}}{\longmapsto} \langle\langle p, \neg p\rangle, \langle p, q\rangle\rangle & \mathcal{RT}_2 \\ & \underset{SAT_{\mathcal{T}}}{\longmapsto} \langle\langle p, q\rangle\rangle & \mathcal{RT}_1 \\ & \underset{SAT_{\mathcal{T}}}{\longmapsto} \langle\langle p, q\rangle\rangle & \mathcal{RT}_5 \end{array}$$

The set of rule schemata proposed could be improved to obtain a more efficient propositional theorem prover from the associated PTS. For example, the rule schemata $\mathcal{RT}_1$ could be mixed with the rule schemata $\mathcal{RT}_2$, $\mathcal{RT}_3$ and $\mathcal{RT}_4$ to avoid occurrences of complementary formulas. Following this idea, we have defined another PTS $\mathcal{T}' = \langle \mathcal{O}_{\mathcal{T}'}, \rightsquigarrow_{\mathcal{T}'}, \models_{\mathcal{T}}\rangle$ associated with the semantic tableaux method in which the propositional objects are lists of formulas without complementary elements and the rule schemata are the following:

$\mathcal{RT}'_1 : \langle\Gamma_1, \neg\neg G, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\rangle$ if $\overline{G} \in \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_2 : \langle\Gamma_1, \neg\neg G, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\langle\Gamma_1, G, \Gamma_2\rangle\rangle$ if $\overline{G} \notin \langle\Gamma_1, \Gamma_2\rangle$

[2] We extend the measure provided in [1] to include equivalence.

$\mathcal{RT}'_3 : \langle\Gamma_1, \alpha, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\rangle$ if $\overline{\alpha_1} \in \langle\Gamma_1, \Gamma_2\rangle$ or $\overline{\alpha_2} \in \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_4 : \langle\Gamma_1, \alpha, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\langle\Gamma_1, \alpha_1, \alpha_2, \Gamma_2\rangle\rangle$ if $\overline{\alpha_1} \notin \langle\Gamma_1, \Gamma_2\rangle$ and $\overline{\alpha_2} \notin \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_5 : \langle\Gamma_1, \beta, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\rangle$ if $\overline{\beta_1} \in \langle\Gamma_1, \Gamma_2\rangle$ and $\overline{\beta_2} \in \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_6 : \langle\Gamma_1, \beta, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\langle\Gamma_1, \beta_1, \Gamma_2\rangle\rangle$ if $\overline{\beta_1} \notin \langle\Gamma_1, \Gamma_2\rangle$ and $\overline{\beta_2} \in \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_7 : \langle\Gamma_1, \beta, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\langle\Gamma_1, \beta_2, \Gamma_2\rangle\rangle$ if $\overline{\beta_1} \in \langle\Gamma_1, \Gamma_2\rangle$ and $\overline{\beta_2} \notin \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_8 : \langle\Gamma_1, \beta, \Gamma_2\rangle \rightsquigarrow_{\mathcal{T}'} \langle\langle\Gamma_1, \beta_1, \Gamma_2\rangle, \langle\Gamma_1, \beta_2, \Gamma_2\rangle\rangle$
if $\overline{\beta_1} \notin \langle\Gamma_1, \Gamma_2\rangle$ and $\overline{\beta_2} \notin \langle\Gamma_1, \Gamma_2\rangle$
$\mathcal{RT}'_9 : \Gamma \rightsquigarrow_{\mathcal{T}'} \mathbf{t}$ if Γ does not have non-literal nor complementary formulas

For this PTS we have considered the same representation function, measure function and model function as we have presented for $\mathcal{T}$. Several computation rules can be considered, taking into account similar considerations than for $\mathcal{T}$.

A final improvement can be considered in both PTSs: the expansion rules can be defined in such a way that duplications in the resulting branches can be avoided.

2.3. Sequents and the Gentzen System

We denote *sequents* as $\Gamma \Rightarrow \Delta$, where Γ and Δ are lists of formulas. An *atomic sequent* is a sequent in which every formula is atomic. A valuation σ makes the sequent $\Gamma \Rightarrow \Delta$ true if and only if $\exists X \in \Gamma(\sigma \not\models X) \vee \exists Y \in \Delta(\sigma \models Y)$; otherwise, we say that σ makes $\Gamma \Rightarrow \Delta$ false. The following are the axiom and rules of Gentzen System G' presented in [10], with two additional rules about equivalence:

$$\frac{}{\Gamma_1, F, \Gamma_2 \Rightarrow \Delta_1, F, \Delta_2}\ (Axiom)$$

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow F, \Delta}{\Gamma_1, \neg F, \Gamma_2 \Rightarrow \Delta}\ (\neg\text{-left}) \qquad \frac{F, \Gamma \Rightarrow \Delta_1, \Delta_2}{\Gamma \Rightarrow \Delta_1, \neg F, \Delta_2}\ (\neg\text{-right})$$

$$\frac{\Gamma_1, F, G, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, F \wedge G, \Gamma_2 \Rightarrow \Delta}\ (\wedge\text{-left}) \qquad \frac{\Gamma \Rightarrow \Delta_1, F, \Delta_2 \quad \Gamma \Rightarrow \Delta_1, G, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \wedge G, \Delta_2}\ (\wedge\text{-right})$$

$$\frac{\Gamma_1, F, \Gamma_2 \Rightarrow \Delta \quad \Gamma_1, G, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, F \vee G, \Gamma_2 \Rightarrow \Delta}\ (\vee\text{-left}) \qquad \frac{\Gamma \Rightarrow \Delta_1, F, G, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \vee G, \Delta_2}\ (\vee\text{-right})$$

$$\frac{\Gamma_1, \Gamma_2 \Rightarrow F, \Delta \quad \Gamma_1, G, \Gamma_2 \Rightarrow \Delta}{\Gamma_1, F \rightarrow G, \Gamma_2 \Rightarrow \Delta}\ (\rightarrow\text{-left}) \qquad \frac{F, \Gamma \Rightarrow \Delta_1, G, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \rightarrow G, \Delta_2}\ (\rightarrow\text{-right})$$

$$\frac{\Gamma_1, F, G, \Gamma_2 \Rightarrow \Delta \quad \Gamma_1, \Gamma_2 \Rightarrow F, G, \Delta}{\Gamma_1, F \leftrightarrow G, \Gamma_2 \Rightarrow \Delta}\ (\leftrightarrow\text{-left})$$

$$(p \to q) \land p \Rightarrow \qquad \frac{(p \to q), p \Rightarrow}{(p \to q) \land p \Rightarrow} \qquad \cfrac{\cfrac{p \Rightarrow p \quad q, p \Rightarrow}{(p \to q), p \Rightarrow}}{(p \to q) \land p \Rightarrow} \qquad \cfrac{\cfrac{\cfrac{}{p \Rightarrow p} \quad q, p \Rightarrow}{(p \to q), p \Rightarrow}}{(p \to q) \land p \Rightarrow}$$

Figure 2. An example of sequent method

$$\frac{F, \Gamma \Rightarrow \Delta_1, G, \Delta_2 \quad G, \Gamma \Rightarrow \Delta_1, F, \Delta_2}{\Gamma \Rightarrow \Delta_1, F \leftrightarrow G, \Delta_2} \; (\leftrightarrow\text{-right})$$

Note that in these rules the symbols Δ and Γ represent lists of formulas and no conditions are imposed about them. Thus, each rule is actually a rule schema that can be instantiated with specific lists of formulas Δs and/or Γs. In particular, the *Axiom* rule represents an infinite set of axioms.

A formula F has a proof in the Gentzen System if the sequent $\Rightarrow F$ can be obtained from the axioms by applying the rules. This proof can be built from the sequent $\Rightarrow F$ by applying the rules in reverse order: in every step a set of unproved sequents is considered (initially the only unproved sequent is $\Rightarrow F$), a rule is chosen to decompose one of these sequents, an instance of the rule conclusion, and it is replaced with the rule premises adequately instantiated. If the set of unproved sequents becomes empty, then the formula F has a proof in the Gentzen System (and therefore it is valid). If an unproved sequent cannot be decomposed with the rules, then the formula F is not valid and this sequent provides a countermodel of it. See [10] for more background about the sequent method.

From a constructive point of view, the Gentzen System can be used to build a countermodel of a formula, whenever this formula is not valid. Therefore, to use this method to build a model of a formula F, the initial sequent must be $\Rightarrow \neg F$ or, equivalently, $F \Rightarrow$. An example with the formula $(p \to q) \land p$ is shown in Figure 2. The initial sequent is $(p \to q) \land p \Rightarrow$. The rule $\land$-left can be used to decompose this sequent, obtaining the sequent $(p \to q), p \Rightarrow$. Next, the rule $\to$-left can be used, obtaining the sequents $p \Rightarrow p$ and $q, p \Rightarrow$. The first one can be solved with the *Axiom* rule, but the second one cannot be decomposed. This last sequent provides a countermodel of the initial sequent (that is, p and q true) and, hence, a model of the formula $(p \to q) \land p$.

Again, this method can be seen as a SAT-prover obtained from a propositional transformation system. We now describe the PTS $\mathcal{S} = \langle \mathcal{O}_\mathcal{S}, \rightsquigarrow_\mathcal{S}, \models_\mathcal{S} \rangle$ associated with the sequent method. In this PTS, $\mathcal{O}_\mathcal{S}$ is the set of sequents (represented as pairs of lists of formulas), $\sigma \models_\mathcal{S} S$ if and only if σ makes S false (note the difference with the

tableaux case, where the distinguished valuations make the branches true), and $\rightsquigarrow_{\mathcal{S}}$ the set of rules given by the following rule schemata:

$\mathcal{RS}_1$: $\langle \Gamma_1, F, \Gamma_2 \rangle \Rightarrow \langle \Delta_1, F, \Delta_2 \rangle \rightsquigarrow_{\mathcal{S}} \langle \rangle$
$\mathcal{RS}_2$: $\langle \Gamma_1, \neg F, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_{\mathcal{S}} \langle \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, \Delta \rangle \rangle$
$\mathcal{RS}_3$: $\langle \Gamma_1, F \wedge G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_{\mathcal{S}} \langle \langle F, G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_4$: $\langle \Gamma_1, F \vee G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_{\mathcal{S}} \langle \langle F, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta, \langle G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_5$: $\langle \Gamma_1, F \rightarrow G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_{\mathcal{S}} \langle \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, \Delta \rangle, \langle G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta \rangle$
$\mathcal{RS}_6$: $\langle \Gamma_1, F \leftrightarrow G, \Gamma_2 \rangle \Rightarrow \Delta \rightsquigarrow_{\mathcal{S}} \langle \langle F, G, \Gamma_1, \Gamma_2 \rangle \Rightarrow \Delta, \langle \Gamma_1, \Gamma_2 \rangle \Rightarrow \langle F, G, \Delta \rangle \rangle$
$\mathcal{RS}_7$: $\Gamma \Rightarrow \langle \Delta_1, \neg F, \Delta_2 \rangle \rightsquigarrow_{\mathcal{S}} \langle \langle F, \Gamma \rangle \Rightarrow \langle \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_8$: $\Gamma \Rightarrow \langle \Delta_1, F \wedge G, \Delta_2 \rangle \rightsquigarrow_{\mathcal{S}} \langle \Gamma \Rightarrow \langle F, \Delta_1, \Delta_2 \rangle, \Gamma \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_9$: $\Gamma \Rightarrow \langle \Delta_1, F \vee G, \Delta_2 \rangle \rightsquigarrow_{\mathcal{S}} \langle \Gamma \Rightarrow \langle F, G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{10}$: $\Gamma \Rightarrow \langle \Delta_1, F \rightarrow G, \Delta_2 \rangle \rightsquigarrow_{\mathcal{S}} \langle \langle F, \Gamma \rangle \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{11}$: $\Gamma \Rightarrow \langle \Delta_1, F \leftrightarrow G, \Delta_2 \rangle \rightsquigarrow_{\mathcal{S}} \langle \langle F, \Gamma \rangle \Rightarrow \langle G, \Delta_1, \Delta_2 \rangle, \langle G, \Gamma \rangle \Rightarrow \langle F, \Delta_1, \Delta_2 \rangle \rangle$
$\mathcal{RS}_{12}$: $\Gamma \Rightarrow \Delta \rightsquigarrow_{\mathcal{S}} \mathbf{t}$ if $\Gamma \Rightarrow \Delta$ is an atomic sequent and $\Gamma \cap \Delta = \emptyset$

These rule schemata correspond to the Gentzen System rules presented above. The rule schemata $\mathcal{RS}_1$ corresponds with the *Axiom* rule. The rule schemata $\mathcal{RS}_2$, $\mathcal{RS}_3$, $\mathcal{RS}_4$, $\mathcal{RS}_5$ and $\mathcal{RS}_6$ correspond to the *left* rules, and the rule schemata $\mathcal{RS}_7$, $\mathcal{RS}_8$, $\mathcal{RS}_9$, $\mathcal{RS}_{10}$ and $\mathcal{RS}_{11}$ to the *right* rules. The rule $\mathcal{RS}_{12}$ checks if a sequent is atomic and the *Axiom* rule cannot be applied to it.

As in the case of the tableaux method, we define a procedure $SAT_{\mathcal{S}}$, obtained from the generic procedure $SAT_{\mathcal{G}}$, by giving concrete computation rule, representation, measure and model functions for the above PTS.

The representation function $i_{\mathcal{S}}$ builds the sequent $F \Rightarrow$ for every $F \in \mathbb{P}(\Sigma)$. Thus, $\sigma \models F \iff \sigma \models_{\mathcal{S}} i_{\mathcal{S}}(F)$ (property $\mathcal{P}_2$).

We consider any computation rule, $r_{\mathcal{S}}$, such that, for every sequent S, $r_{\mathcal{S}}(S)$ is the result of applying one of the above rule schemata to S, whenever such rule could be applied. We can consider several computation rules representing different versions of the sequent method, depending on the preference order considered in the rule schemata.

We define the measure function $\mu_{\mathcal{S}}$ as the number of occurrences of propositional connectives in a sequent. The expansion rules reduce this number, therefore $S_i \in r_{\mathcal{S}}(S) \Longrightarrow \mu_{\mathcal{S}}(S_i) < \mu_{\mathcal{S}}(S)$ (property $\mathcal{P}_1$).

Given an expansion rule $S \rightsquigarrow_{\mathcal{S}} L$ with $L \neq \mathbf{t}$, it can be easily proved that $\sigma \models_{\mathcal{S}} S \iff \exists S_i \in L, \sigma \models_{\mathcal{S}} S_i$. By our definition of computation rule, this implies property $\mathcal{P}_3$.

Finally, we define the model function $\gamma_{\mathcal{S}}$ such that for every atomic non-axiom sequent S, $\gamma_{\mathcal{S}}(S) \models p$ if and only if p occurs in the left part of S. Obviously, if $r_{\mathcal{S}}(S) = \mathbf{t}$ then $\gamma_{\mathcal{S}}(S) \models_{\mathcal{S}} S$ (property $\mathcal{P}_4$).

Then, by Theorem 1, the algorithm $SAT_{\mathcal{S}}$ terminates for any formula and is complete and sound. The algorithm applied to the example of Figure 2 performs the following steps (represented as $\underset{SAT_{\mathcal{S}}}{\longmapsto}$):

$$\begin{aligned}
\langle\langle(p \rightarrow q) \wedge p\rangle \Rightarrow\rangle &\xrightarrow[SAT_S]{} \langle\langle p \rightarrow q, p\rangle \Rightarrow\rangle && \mathcal{RS}_3 \\
&\xrightarrow[SAT_S]{} \langle p \Rightarrow p, \langle q, p\rangle \Rightarrow\rangle && \mathcal{RS}_5 \\
&\xrightarrow[SAT_S]{} \langle\langle q, p\rangle \Rightarrow\rangle && \mathcal{RS}_1 \\
&\xrightarrow[SAT_S]{} \langle\langle q, p\rangle \Rightarrow\rangle && \mathcal{RS}_{12}
\end{aligned}$$

The set of rule schemata proposed could be improved to obtain a more efficient propositional theorem prover from the associated PTS. We could consider similar ideas as the presented for the tableaux method. For example, the rule schemata $\mathcal{RS}_1$ could be mixed with the rule schemata $\mathcal{RS}_2$ to $\mathcal{RS}_{11}$ avoiding occurrences of the same formula in both sides of a sequent. The expansion rules also could be defined avoiding repetitions of formulas in the left side or the right side of a sequent.

2.4. Davis–Putnam–Logeman–Loveland method

The Davis–Putnam–Logeman–Loveland (DPLL for short) method [5, 6] is a procedure to decide the satisfiability of a set of clauses. Thus, if $\mathcal{FC}$ is a procedure to obtain a set of clauses logically equivalent to a formula F, we can use the DPLL method to decide the satisfiability of F, applying the method to $\mathcal{FC}(F)$.

The basic operation in this method is the reduction of a set of clauses with respect to a literal. In this operation a literal L appearing in a clause is assumed to be true and then the set is reduced accordingly, removing the clauses in which L occurs and removing the literal $\overline{L}$ from the clauses in which it appears. More precisely, given a set of clauses S and a literal L, we define $S_L = \{C - \{\overline{L}\} : C \in S \text{ and } L \notin C\}$.

Roughly speaking, the DPLL method starts with an initial set of clauses and reduces the problem of checking its satisfiability to check the satisfiability of some of its reductions by a given literal. In every step a set of clauses S is considered, and one of the following actions is performed:

- If S is empty, then it is satisfiable and the set of literals used in the reduction process to obtain S from the original set of clauses is a model of the original set of clauses.
- If S contains the empty clause, then it is removed and the next set of clauses is considered.
- If S contains an unitary clause $\{L\}$, then S is removed and the set of clauses S_L is considered.
- Otherwise, a literal L occurring in some clause of S is chosen, S is removed and the sets of clauses S_L and $S_{\overline{L}}$ are considered.

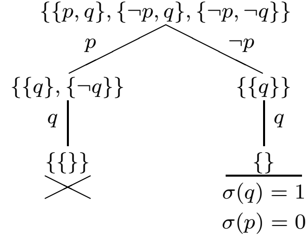

Figure 3. An example of DPLL method

This process is iterated until one of the sets obtained is empty (in which case the original set of clauses is satisfiable) or until all the sets finally obtained contain the empty clause (and then the initial set of clauses is unsatisfiable). An example with the set of clauses $S = \{\{p,q\},\{\neg p,q\},\{\neg p,\neg q\}\}$ is shown in Figure 3. Initially, the literal p is chosen and S is replaced with $S_p = \{\{q\},\{\neg q\}\}$ and $S_{\neg p} = \{\{q\}\}$. S_p contains the unitary clause $\{q\}$, and the new set of clauses is $S_{p,q} = \{\{\}\}$, which contains the empty clause and therefore it is removed. $S_{\neg p}$ contains the unitary clause $\{q\}$, and the new set of clauses is $S_{\neg p,q} = \{\}$. This last set of clauses is satisfiable, and the literals used in the reduction process provide a model of S.

We now present a propositional transformation system reflecting the Davis-Putnam method. First, it must be noticed that the set of literals used in the reduction process is needed to build a model of the original set of clauses. Thus, the objects must be pairs $\langle S, M\rangle$ where S is a set of clauses and M is the set of literals used in the reduction process to obtain S. Therefore, M cannot contain complementary literals and for every L in M, neither L nor $\overline{L}$ is in some clause in S. We denote $\mathcal{O}_{\mathcal{D}}$ this set of objects.

Second, note that the reduction process finishes when the object $\langle\langle\rangle, M\rangle$ is obtained. In this case M is the set of literals used in the reduction process to obtain the empty set of clauses. Thus, M provides a model of the initial set of clauses. Then, the distinguished valuations of $\langle\langle\rangle, M\rangle$ must be models of every literal in M. In addition, to ensure property $\mathcal{P}_2$, the distinguished valuations of the initial object $\langle S, \langle\rangle\rangle$ must be models of every clause in S. Taking into account these considerations, we say that a valuation σ is a distinguished valuation of an object $\langle S, M\rangle$ if and only if σ is a model of every clause in S and every literal in M, and we define $\sigma \models_{\mathcal{D}} \langle S, M\rangle$ if and only if $\langle S, M\rangle \in \mathcal{O}_{\mathcal{D}}$ and σ is a distinguished valuation of $\langle S, M\rangle$.

Finally, the PTS associated with the DPLL method is $\mathcal{D} = \langle \mathcal{O}_{\mathcal{D}}, \leadsto_{\mathcal{D}}, \models_{\mathcal{D}} \rangle$. Where $\mathcal{O}_{\mathcal{D}}$ and $\models_{\mathcal{D}}$ are as described above and $\leadsto_{\mathcal{D}}$ is the relation given by the following rule schemata:

$\mathcal{RD}_1 : \langle S, M \rangle \leadsto_{\mathcal{D}} \langle\rangle$ if the empty clause is in S
$\mathcal{RD}_2 : \langle S, \langle L_1, \ldots, L_n \rangle \rangle \leadsto_{\mathcal{D}} \langle \langle S_L, \langle L, L_1, \ldots, L_n \rangle \rangle \rangle$
if the unitary clause $\{L\}$ is in S
$\mathcal{RD}_3 : \langle S, \langle L_1, \ldots, L_n \rangle \rangle \leadsto_{\mathcal{D}} \langle \langle S_L, \langle L, L_1, \ldots, L_n \rangle \rangle, \langle S_{\overline{L}}, \langle \overline{L}, L_1, \ldots, L_n \rangle \rangle \rangle$
where L is a literal in a clause in S
$\mathcal{RD}_4 : \langle \langle\rangle, M \rangle \leadsto_{\mathcal{D}} \mathbf{t}$

where $\langle S, M \rangle$ and $\langle S, \langle L_1, \ldots, L_n \rangle \rangle$ are elements in $\mathcal{O}_{\mathcal{D}}$.

As in the previous subsections, we define the functions needed to define the procedure $SAT_{\mathcal{D}}$ as an instance of the generic procedure $SAT_{\mathcal{G}}$. In this case, the representation function $i_{\mathcal{D}}$ builds a pair $\langle \mathcal{FC}(F), \langle\rangle \rangle$, where $\mathcal{FC}$ is assumed to be a correct procedure to obtain a set of clauses logically equivalent to F; that is, $\sigma \models F \iff \sigma \models \mathcal{FC}(F) \iff \sigma \models_{\mathcal{D}} i_{\mathcal{D}}(F)$ (property $\mathcal{P}_2$).

We consider a computation rule, $r_{\mathcal{D}}$, that applies the expansion rules schemata in the following preference order $\mathcal{RD}_1$, $\mathcal{RD}_4$, $\mathcal{RD}_2$ and $\mathcal{RD}_3$. This order reduces the number of reduction steps. First, we try to identify the dead ends, using the rule $\mathcal{RD}_1$, and the sucessful ends, using the rule $\mathcal{RD}_4$. Then, we try to simplify the objects using the rule $\mathcal{RD}_2$ because this does not produce bifurcations. Finally, we apply the rule $\mathcal{RD}_3$. Anyway, any other order of application of these rules could be used.

We define the measure function $\mu_{\mathcal{D}}$, such that, for every $\langle S, M \rangle \in \mathcal{O}_{\mathcal{D}}$, $\mu_{\mathcal{D}}(\langle S, M \rangle)$ is the total number of literals of the clauses of S. The expansion rules reduce this value, therefore $\langle S_i, M_i \rangle \in r_{\mathcal{D}}(\langle S, M \rangle) \implies \mu_{\mathcal{D}}(\langle S_i, M_i \rangle) < \mu_{\mathcal{D}}(\langle S, D \rangle)$ (property $\mathcal{P}_1$).

Given an expansion rule $\langle S, M \rangle \leadsto_{\mathcal{D}} L$ with $L \neq \mathbf{t}$, it can be easily proved that $\sigma \models_{\mathcal{D}} \langle S, M \rangle \iff \exists \langle S_i, M_i \rangle \in L, \sigma \models_{\mathcal{D}} \langle S_i, M_i \rangle$. By our definition of computation rule, this implies property $\mathcal{P}_3$.

Finally, we define the model function $\gamma_{\mathcal{D}}$ such that for every pair $\langle \langle\rangle, M \rangle$ and $p \in \Sigma$, $\gamma_{\mathcal{D}}(\langle \langle\rangle, M \rangle) \models p$ if and only if $p \in M$. Obviously, if $r_{\mathcal{D}}(\langle S, M \rangle) = \mathbf{t}$ then $S = \langle\rangle$ and $\gamma_{\mathcal{D}}(\langle S, M \rangle) \models_{\mathcal{D}} \langle S, M \rangle$ (property $\mathcal{P}_4$).

Then, by Theorem 1, the algorithm $SAT_{\mathcal{D}}$ terminates for any formula and is complete and sound. The algorithm applied to the formula $(p \to q) \wedge p$ performs the following steps (represented as $\underset{SAT_{\mathcal{D}}}{\longmapsto}$):

$$
\begin{array}{llr}
\langle \langle \langle \{\neg p, q\}, \{p\} \rangle, \langle\rangle \rangle \rangle & \underset{SAT_{\mathcal{D}}}{\longmapsto} \langle \langle \langle \{q\} \rangle, \langle p \rangle \rangle \rangle & \mathcal{RD}_3 \\
& \underset{SAT_{\mathcal{D}}}{\longmapsto} \langle \langle \langle\rangle, \langle q, p \rangle \rangle \rangle & \mathcal{RD}_2 \\
& \underset{SAT_{\mathcal{D}}}{\longmapsto} \langle \langle \langle\rangle, \langle q, p \rangle \rangle \rangle & \mathcal{RD}_4
\end{array}
$$

Again, the set of rule schemata proposed could be improved to obtain a more efficient propositional SAT prover from the associated

PTS. For example, the rule $\mathcal{R}_{\mathcal{D}4}$ could be changed to detect the end of the reduction process when a set of clauses S only has pure literals (those that only appear positive or negative in the set of clauses) and the rule $\mathcal{R}_{\mathcal{D}3}$ could be changed to choose only non-pure literals.

3. Formalizing the generic SAT-prover in ACL2

Now we describe a tool based on the theoretical development presented in Subsection 2.1. This tool builds a certified propositional theorem prover from a Propositional Transformation System and its associated functions as it was described in Algorithm 1, whenever the properties $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ and $\mathcal{P}_4$ are satisfied. It is built on top of the ACL2 system. In this section, we show how the generic development of Subsection 2.1 is formalized in ACL2.

3.1. A brief introduction to ACL2

ACL2 is a programming language, a logic for formal reasoning about programs defined in the programming language, and a theorem prover supporting mechanized reasoning in the logic. It is developed by J Moore and Matt Kaufmann in the University of Texas at Austin, considered as an "industrial-strength" successor of Nqthm, also known as the Boyer-Moore theorem prover.

As a programming language, ACL2 is an extension of a subset of Common Lisp, containing most of the applicative part of that language. The ACL2 logic is a quantifier-free, first-order logic with equality, describing the functions of the programming language. The syntax of terms is that of Common Lisp and the logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference of the logic include those for propositional calculus, equality and instantiation.

One important rule of inference is the *principle of induction*, that permits proofs by well-founded induction on the ordinal ε_0. The theory has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers, given by the predicate `e0-ordinalp` and the order `e0-ord-<`.

By the *principle of definition* (using `defun`), new function definitions are admitted as axioms only if there exists a measure in which the arguments of each recursive call decrease with respect to a well-founded relation, ensuring in this way that no inconsistencies are introduced by new definitions.

Some higher order functionality is provided by means of the `encapsulate` mechanism [13] which allows the user to introduce new

function symbols by axioms constraining them to have certain properties (to ensure consistency, a witness local function having the same properties has to be exhibited). Inside an `encapsulate`, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms. This mechanism behaves like an universal quantifier over a set of functions abstractly defined with it.

A derived rule of inference, called *functional instantiation*, gives some features of a higher order logic by allowing to instantiate the function symbols of a previously proved theorem, replacing them with other function symbols or lambda expressions, provided it can prove that the replacements satisfy the constraints on the old symbols.

The ACL2 theorem prover mechanizes the logic. The prover is mainly based on applying simplification and induction. Roughly speaking, when the prover tries to prove a conjecture, it simplifies the formula. If it obtains `t`, then the conjecture is proved. Otherwise, it guesses an (often suitable) induction scheme, and recursively tries to prove the subgoals generated.

The theorem prover is automatic in the sense that once submitted a conjecture (by the command `defthm`), the user can no longer interact with the system. But in a wider sense, the prover is interactive: non-trivial results often fail to be proved unless the user previously proves lemmas that can be used in subsequent proofs as rewriting rules. In this way, the user can help the prover to find a preconceived hand proof. This is the way we have interacted with the system to obtain the results presented in this section. For a detailed description of ACL2, we refer the reader to the ACL2 book [11].

3.2. Definition of the generic algorithm

The first step to reason in ACL2 about the algorithm $SAT_{\mathcal{G}}$, is to define in the ACL2 logic the functions introduced by the generic framework presented in Section 2.1. The names of these ACL2 functions and their intended meanings are shown in the following table:

`gen-object-p`(O)	$O \in \mathcal{O}$
`gen-repr`(F)	$i(F)$
`gen-comp-rule`(O)	$r(O)$
`gen-dist-val`(σ, O)	$\sigma \models_{\mathcal{G}} O$
`gen-model`(O)	$\gamma(O)$
`gen-measure`(O)	$\mu(O)$
`gen-select`(lst)	selects an element from a list lst

These functions are not introduced in the ACL2 logic using the principle of definition. Since they are generic, we define them by means

of the `encapsulate` mechanism, constraining them to have certain properties[3]. In this case, the properties about the generic functions are the following[4]:

Assumption: `gen-object-p-gen-repr`
`propositional-p`$(F) \rightarrow$ `gen-object-p`(`gen-repr`(F))

Assumption: `gen-object-p-gen-comp-rule`
`gen-object-p`$(O_1) \wedge (O_2 \in$ `gen-comp-rule`$(O_1))$
$\rightarrow$ `gen-object-p`(O_2)

Assumption: `e0-ordinalp-gen-measure`
`e0-ordinalp`(`gen-measure`(O))

Assumption: `P1`
$O_2 \in$ `gen-comp-rule`(O_1)
$\rightarrow$ `gen-measure`$(O_2) <$ `gen-measure`(O_1)

Assumption: `P2`
`propositional-p`(F)
$\rightarrow$ (`gen-dist-val`$(\sigma,$ `gen-repr`$(F)) \leftrightarrow$ `models`(σ, F))

Assumption: `P3`
`gen-object-p`$(O) \wedge$ (`gen-comp-rule`$(O) \neq$ **t**)
$\rightarrow$ (`gen-dist-val`(σ, O)
$\leftrightarrow$ `gen-dist-val-list`$(\sigma,$ `gen-comp-rule`$(O))$)

Assumption: `P4`
`gen-object-p`$(O) \wedge$ (`gen-comp-rule`$(O) =$ **t**)
$\rightarrow$ `gen-dist-val`(`gen-model`$(O), O)$

Assumption: `gen-select-member`
`consp`$(lst) \rightarrow$ (`gen-select`$(lst) \in lst$)

The first three properties state that the functions `gen-repr`, `gen-comp-rule` and `gen-measure` take values as expected, when acting on elements of their intended domains. The properties named `P1`, `P2`, `P3` and `P4` are the corresponding formalization of the properties $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$ and $\mathcal{P}_4$, respectively, as defined in the hypothesis of Theorem 1.

[3] The local witnesses are irrelevant to the definition of the generic algorithm and the proof of its properties, so we omit them here.

[4] The expressions provided to ACL2 are written in Common Lisp notation but, to improve their legibility, we present them here using a “infix” notation.

The functions `propositional-p` and `models` are defined in a previous ACL2 formalization about the syntax and semantics of propositional logic; they define, respectively, the propositional formulas and models of formulas. The function `gen-dist-val-list` can be seen as a generalized disjunction of the predicate `gen-dist-val` acting on the objects of a list. The symbol $<$ denotes the "less than" relation between ordinals. Finally, note that we also introduce a function `gen-select`, that selects an element from any non-empty list. This function is needed in the definition of the generic SAT algorithm.

Once the functions of our generic framework have been introduced, we define in ACL2 the function `generic-sat`, implementing the algorithm $SAT_{\mathcal{G}}$:

DEFINITION:
`generic-sat-lst`(*O-lst*) =
  if `endp`(*O-lst*) **then nil** (1)
  else let* *O* **be** `gen-select`(*O-lst*), (2)
    rest **be** `remove-one`(`gen-select`(*O-lst*), *O-lst*),
    expansion **be** `gen-comp-rule`(*O*) (3)
   in if *expansion* = **t then** `list`(*O*) (4)
    else `generic-sat-lst`(*expansion* @ *rest*)
MEASURE: `gen-measure-lst`(*O-lst*)
WELL FOUNDED RELATION: $<_{mul}$

DEFINITION:
`generic-sat`(F) = `generic-sat-lst`(`list`(`gen-repr`(F)))

where the symbol @ is the "append" operation between lists.

Note that the main function of this algorithm is given by the recursive function `generic-sat-lst`, acting on a list of objects to be expanded. This function implements the while loop in the definition of $SAT_{\mathcal{G}}$. The termination of this loop is justified by the measure `gen-measure-lst`(*O-lst*) and the multiset well-founded relation $<_{mul}$. We will explain more about this issue in the next subsection.

When a rule of the form $\langle O, \mathbf{t} \rangle$ is applied to a selected object O, the algorithm returns a singleton list containing O (4). According to the property assumed about the function `gen-model`, this object has a distinguished valuation. Thus, returning the object is useful to provide a model of the input formula. On the other hand, when there are no more objects to be expanded, the algorithm returns **f**, represented as the ACL2 symbol `nil` (1).

This algorithm is left unspecified in two aspects: first, no concrete computation rule is defined by the generic function `gen-comp-rule` (3);

second, the object to which the expansion rule is applied, selected by the abstractly defined function `gen-select`, is not specified (2).

3.3. Termination

As it was pointed out in Subsection 3.1, new function definitions are admitted in ACL2 only if there exists a well-founded measure in which the arguments of each recursive call decrease. In the case of the function `generic-sat-lst` the heuristics of ACL2 are not able to find a suitable termination argument, so we must explicitly provide a measure on its argument an show that this measure decreases in every recursive call with respect to a well-founded relation.

The only predefined well-founded relation in ACL2 is `e0-ord-<`, implementing the usual order between ordinals less than ε_0. The function `e0-ordinalp` recognizes those ACL2 objects representing such ordinals. If we want to define a new well-founded relation in ACL2, we have to explicitly provide a monotone ordinal function, and prove the corresponding order-preserving theorem (see [11] for details).

To show termination of `generic-sat-lst`, we follow the lines described in the informal proof given in Section 2.1. The measure associated to its argument is given by a function `gen-measure-lst` that computes the list of the ordinal measures of the objects of a given list. This measure decreases with respect to the multiset relation induced by `e0-ord-<`.

Since `e0-ord-<` is well-founded, so is its induced multiset relation [7]. A formal proof of the well-foundedness of the multiset relation induced by given well-founded relation was formalized in the ACL2 logic in [17], where the `defmul` tool was also developed. This tool automatically generates the definitions and prove the theorems needed to introduce in ACL2 the multiset relation induced by a given well-founded relation. In our case, we only need the following `defmul` call:

```
(defmul (e0-ord-< nil e0-ordinalp e0-ord-<-fn nil nil))
```

This automatically generates the definition of `mul-e0-ord-<`, (denoted as $<_{mul}$ in the following), implementing the multiset relation on finite multisets (lists) of ordinals induced by the relation `e0-ord-<`. And it also automatically proves the theorems needed to introduce this relation as a well-founded relation in ACL2. See details about the `defmul` syntax in [17].

The main termination property of `generic-sat-lst` is given by the following theorem, establishing that the measure `gen-measure-lst` decreases in every recursive call with respect to the well-founded relation $<_{mul}$:

THEOREM: generic-sat-lst-termination-property
let* O **be** `gen-select`(O-lst),
$rest$ **be** `remove-one`(`gen-select`(O-lst), O-lst),
$expansion$ **be** `gen-comp-rule`(O)
in `consp`(O-lst) $\wedge$ ($expansion \neq$ **t**)
$\rightarrow$ `gen-measure-lst`($expansion$ @ $rest$)
$<_{mul}$ `gen-measure-lst`(O-lst)

Having proved this theorem (and given that $<_{mul}$ is well-founded, as it was automatically proved by the above call to `defmul`) the definition of `generic-sat-lst` is shown to be terminating and it is admitted in the logic (and therefore, the definition of `generic-sat`).

3.4. Soundness and completeness

The following theorems establish the formal properties of the function `generic-sat` (soundness and completeness):

THEOREM: soundness-generic-sat
`propositional-p`(F) $\wedge$ `generic-sat`(F)
$\rightarrow$ `models`(`generic-mod`(F), F)

THEOREM: completeness-generic-sat
`propositional-p`(F) $\wedge$ `models`(σ, F) $\rightarrow$ `generic-sat`(F)

Due to the lack of existential quantification in the ACL2 logic, the soundness theorem has to be formulated by explicitly giving a model of the formula F. This model can be easily obtained from the result returned by the `generic-sat` procedure, as defined by the function `generic-mod`:

DEFINITION:
`generic-mod`(F) =
if `consp`(`generic-sat`(F))
then `gen-model`(`first`(`generic-sat`(F)))
else nil

The above two theorems formalize Theorem 1 in ACL2. They are proved along the lines of the informal proof given in Section 2.1, basically first proving by induction analogous properties about `generic-sat-lst`. Of course, the properties assumed about the generic functions showed in the Subsection 3.2 play a crucial role. See details of the mechanical proof in [16].

4. Instantiating the generic framework

Concrete SAT-provers will be given by defining concrete counterparts of the abstractly defined functions given in Subsection 3.2. With these concrete functions, one can define concrete versions of the algorithm `generic-sat`.

We can also obtain concrete versions of the termination, soundness and completeness theorems: if the assumed properties about the generic functions are verified by the concrete functions, then by functional instantiation we can easily conclude termination, soundness and completeness of the concrete SAT-prover.

4.1. An overview of the instantiation process

We describe in this section how we perform the instantiation process in order to obtain a certified specific SAT-prover as a concrete instantiation of the generic framework. First of all, we need a concrete version of the generic functions given in Subsection 3.2. Let us assume, for example, that we have a PTS such that its associated functions are given by functions named `object-p`, `repr`, `comp-rule`, `dist-val`, `model`, `measure` and `select`, concrete counterparts of the generic functions defined in Subsection 3.2, and reflecting the given PTS. We also need the functions `dist-val-list`, a generalized disjunction of the predicate `dist-val` over a list of objects, and `object-list-p`, a recognizer for proper (null terminated) lists of objects.

The following steps would have to be performed in order to obtain a certified SAT-prover:

1. The above concrete counterparts of the generic functions have to be defined in ACL2. We will assume that these functions are executable (that is, they are not defined via `encapsulate`).

2. Concrete versions of the assumed properties about the generic functions (given in Subsection 3.2) have to be proved.

3. The concrete counterparts of the derived functions (with the final goal of defining the concrete version of the function `generic-sat`), have to be defined. Note that these functions will be executable.

4. Finally, concrete versions of the termination, soundness and completeness theorems have to be formulated and proved by functional instantiation from the generic theorems.

The same procedure would have to be done for every concrete instantiation of the generic framework, so it makes sense to use a tool to

mechanize this process to some extent. In particular, the last two steps can be completely automated.

In [14], we describe a user tool that we developed to instantiate generic ACL2 theories. This tool turns out to be a valuable help in this context, where we have developed a generic theory about SAT-provers and we want to instantiate the theory to obtain concrete, formally verified and executable SAT-provers.

This tool mainly consists of a macro named `def-generic-theory`, which receives as argument a *string* identifying the theory and a sequence of ACL2 events (definitions and theorems), some of which are labeled to be instantiated. When an ACL2 book[5] developing a generic theory is created, we include a call to this macro. The effect of the macro call is to define another macro that automatically builds concrete events as instances of the generic events, and to instruct the prover to establish the generated theorems by functional instantiation of the generic ones (thus, they are automatically proved).

For example, in the book that formalizes the generic framework for SAT-provers (as described in the previous section), we include the following:

```
(def-generic-theory *generic-sat*
  <events>)
```

Here `<events>` is a sequence containing the events corresponding to the generic definitions and theorems that can be instantiated by other ACL2 books. In particular, the definition of `generic-sat` and the theorems establishing its properties. When this macro call is executed, it defines a new macro that receives as input a functional substitution, generates the corresponding functional instantiation of the instantiable events.

For example, once the functions implementing the concrete counterparts of the generic functions are defined and we have proved that they verify the assumed properties, we include the book with the generic SAT-prover formalization. At that point, a macro `definstance-*generic-sat*` is automatically defined, and we can use this macro to automatically generate instantiated events for the concrete SAT-prover, as follows:

```
(definstance-*generic-sat*
  ((gen-object-p       object-p)
```

[5] A collection of ACL2 definitions and proved theorems is usually stored in a certified file of *events* (a *book* in the ACL2 terminology), that can be included in other books.

```
(gen-object-list-p  object-list-p)
(gen-repr           repr)
(gen-dist-val       dist-val)
(gen-dist-val-list  dist-val-list)
(gen-comp-rule      comp-rule)
(gen-select         select)
(gen-measure        measure)
(gen-model          model))
"-concrete")
```

Note that this macro receives as input a functional substitution, associating every function of the generic framework with its concrete counterpart. Note that the functions `object-list-p` and `dist-val-list` must also be included. It also receives a string, used to name the new events generated, by appending it to the name of the original event. For example, in the above call, we used the prefix `"-concrete"`.

The result of this macro call is the *automatic* generation of the events needed to define and verify in ACL2 the concrete SAT-prover. As a consequence, the definition of a function named `generic-sat-concrete` is generated, as a functional instance of `generic-sat`. And also the following theorems, establishing the soundness and completeness of `generic-sat-concrete`, are automatically generated and proved:

THEOREM: soundness-generic-sat-concrete
  $\texttt{propositional-p}(F) \wedge \texttt{generic-sat-concrete}(F)$
    $\rightarrow \texttt{models}(\texttt{generic-mod-concrete}(F), F)$

THEOREM: completeness-generic-sat-concrete
  $\texttt{propositional-p}(F) \wedge \texttt{models}(\sigma, F)$
    $\rightarrow \texttt{generic-sat-concrete}(F)$

Note that, once the concrete counterparts of the generic functions verifying the properties showed in Subsection 3.2 are proved, no additional interactive proof effort is needed to define and verify the concrete and executable SAT-prover.

4.2. A TABLEAUX BASED SAT-PROVER

Along the lines of Subsection 2.2, we have defined in ACL2 several tableaux based instantiations of the generic framework. For that purpose we have defined a tableaux version of the generic functions given in Subsection 3.2: `tableaux-object-p`, `tableaux-repr`,

`tableaux-comp-rule`, `tableaux-dist-val`, `tableaux-model`, `tableaux-measure` and `tableaux-select`.

For the first tableaux based SAT-prover, these functions are defined as suggested in Subsection 2.2. For example, the definition of the computation rule is the following (recall that in this case, objects are lists of propositional formulas, representing branches in a tableau):

DEFINITION:

```
tableaux-comp-rule(θ) =
   if closed-tableau(θ) then nil                                    RT1
   else let F be one-formula(θ)
            θ' be remove(F,θ)
        in if doubly-neg-p(F)
              then list(add(neg-neg-component(F),θ'))               RT2
           elseif alfa-formula-p(F)
              then list(add(component-1(F),
                            add(component-2(F),θ'))))               RT3
           elseif beta-formula-p(F)
              then list(add(component-1(F),θ')),
                        add(component-2(F),θ')))                    RT4
           else t                                                   RT5
```

Here the function `closed-tableau` checks if a branch has complementary formulas. In this case, the empty list is returned. Otherwise, a formula is selected using the function `one-formula`, and the branch is expanded according to the type of the formula selected, as described by the rules $\leadsto_{\mathcal{T}}$.

Note that this computation rule implements a strategy for applying the tableaux expansion rules in a preference order. This order is implicitly given by the function `one-formula`. Any other strategy could have been defined, provided that the properties assumed about the generic functions could be proved for the concrete counterparts. In this case, these properties are proved easily, except for `P3` and `P4`, which are somewhat more elaborate.

Once the assumed properties in the generic framework have been proved for the tableaux case, we can automatically instantiate the generic SAT-prover algorithm as we have described in the previous subsection. As a result, we obtain a certified function implementing the algorithm $SAT_{\mathcal{T}}$ discussed in Section 2.2.

We have also considered the improved PTS $\mathcal{T}'$ presented in the last paragraphs of Section 2.2. In this case objects are lists of formulas without complementary elements and the computation rule is defined applying the transformations of $\leadsto_{\mathcal{T}'}$ in the order presented in Section

2.2. The SAT-prover obtained is more efficient than the one based on the PTS $\mathcal{T}$.

4.3. Sequent and DPLL based SAT-prover

We follow an analogous procedure to define and verify, sequent and DPLL instantiations of the generic SAT-prover. This is done by a macro call similar to that used in the tableaux case.

As with tableaux, the functional substitution used in the macro call relates the generic functions with their concrete counterparts. Of course, these concrete functions have to be previously defined, their properties proved and the book with the generic development included. These functions are defined as suggested in Subsections 2.3, for the sequent based SAT-prover, and 2.4, for the DPLL based SAT-prover.

4.4. Executability

Functions in the ACL2 logic are total. This means that according to its logical meaning, every function in the logic returns a value for every input. Nevertheless, the ACL2 system provides a mechanism to specify the intended domain of a function by means of logical formulas, called "guards". Although this specification is actually ignored by the logic, the guard verification mechanism allows to evaluate the function directly in Common Lisp: if the guards of a function are verified, then it is ensured that when the function is evaluated on arguments satisfying its guard, then all subsequent function calls during that evaluation will be on arguments satisfying the guard of the called function. The proof obligations generated by the guard verification mechanism ensure this property. Since the primitive Common Lisp functions of ACL2 has guards consistent with the Common Lisp specification, an ACL2 function with its guards verified is Common Lisp compliant and can be evaluated, on arguments satisfying its guard, directly in the underlying Common Lisp.

The main properties about the generic functions needed in the proof obligations generated by the guard verification mechanism have been included in the generic theory. In this way, the instantiation process builds the corresponding properties in the concrete cases, automatizing the guard verification process. Since the instantiation process verifies the guards of the functions implementing the concrete SAT-prover, they are executable in any compliant Common Lisp (with the appropriate ACL2 files loaded).

Table I. Times to solve the N-queens problem

N	Tableaux $\mathcal{T}$	Tableaux $\mathcal{T}'$	Sequent	DPLL
2	0.000	0.000	0.000	0.000
3	0.030	0.030	0.010	0.000
4	0.250	0.230	0.090	0.010
5	1.200	1.070	0.430	0.040
6	106.040	99.190	36.690	0.140
7	375.670	364.460	129.020	0.230

In Table I we present some results of applying the verified SAT-provers to the N-queens problem[6].

The performance of these SAT-provers is far from the results that can be obtained by using any of the current state-of-art SAT-provers. Nevertheless, our emphasis in this paper is not on eficiency, but on obtaining formally verified implementations that can be executed in a widely used programming language like Common Lisp. Of course, efficiency is important and in fact it is our next goal: more efficient SAT-provers could be obtained by using more efficient data structures and techniques, and a formal proof of its correctness could be obtained by proving equivalence theorems with the implementations presented here.

5. Conclusions and further work

We have presented an application of the ACL2 system to the formal verification of a family of SAT decision procedures. That is:

- We have introduced an abstract framework where we describe the essential properties of a family of SAT provers. As a result, we have defined a generic SAT prover and proved its termination, soundness and completeness.

- This abstract framework has been formalized and its main properties proved in the ACL2 system.

[6] All results are in seconds of user CPU on a AMD Athlon(tm) XP 2200+

- We have shown how the abstract framework can be functionally instantiated for a number of concrete SAT provers. This instantiation process can be automated to some extent.
- From these concrete instances, we have obtained compliant Common Lisp executable and formally verified SAT provers.

Since SAT provers are a core component in many practical applications of automated deduction and artificial intelligence, we think that formal verification of executable SAT provers is interesting. As N. Shankar points out in [18], verification of "little proof engines" is one of the challenges for automated reasoning in mathematically rich domains. This work can be seen as a first approach to that goal.

We also think that the work presented here is a good example of the formalization of a generic theory in ACL2 and how a generic development can be instantiated to obtain a number of concrete and executable instances. It is also worth pointing that this work motivated the definition of a tool [14] in ACL2 that can be used to automatize the instantiation of generic ACL2 theories.

As for the proof effort, the following table summarizes the number of definitions, theorems and hints needed to formalize and prove each section. The last column includes information about the number of theorems that need non-trivial hints from the user (we do not count hints for enabling or disabling previous lemmas). This data and the number of theorems, give us an idea of the automation degree of the proofs.

Table II. Proof effort

Section	Definitions	Theorems	Hints
Generic algorithm	16	45	9
Uniform notation	10	15	0
Tableaux based SAT-prover $\mathcal{T}$	13	40	9
Tableaux based SAT-prover $\mathcal{T}'$	16	53	9
Sequent based SAT-prover	20	44	5
Clauses and $\mathcal{FC}$ procedure	26	64	6
DPLL SAT-prover	22	48	6

As for the human time required to complete the whole development presented here, we needed about two years of partial dedication.

Initially we developed independent SAT provers based on semantic tableaux and sequents. After that, we realized that the common structure of both methods could be generalized. This suggested our development of the generic instantiation tool, presented at the third ACL2 workshop [14]. Finally, the generic framework for SAT-provers was presented in the LOPSTR conference [15]. As we have noted in the introduction, this work is an extended and revised version of this paper.

There is some related work in mechanical verification of SAT-provers. A classical example is Boyer and Moore's propositional tautology checker [2], presented as an `IF-THEN-ELSE` normalization procedure and verified using Nqthm (the predecessor of ACL2). This example has been formalized in other systems as well. A more recent work is done by Caldwell [4] using Nuprl and program extraction to obtain a mechanically verified sequent proof system for propositional logic.

The methodology we have followed turns out to be suitable for mechanical verification. Reasoning first about the generic algorithm allows us to concentrate on the essential aspects of the process, making verification tasks easier. Functional instantiation allows us to verify concrete instances of the algorithm, without repeating the main proof effort and allowing some kind of mechanization of the process.

As pointed out at the end of Subsection 4.4, an additional step in this methodology could be refinement. We could define more efficient functions and obtain their properties by proving equivalence theorems with the less efficient ones. In particular, ACL2 allows the use of efficient data structures by means of *single-threaded objects*, which implement destructive operations with an applicative semantics [3]. Our future work will follow this line: using single-threaded objects to implement efficient and verified SAT provers.

References

1. M. Ben-Ari. *Mathematical Logic for Computer Science.* Springer–Verlag, 2001.
2. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
3. R. S. Boyer and J S. Moore. *Single-threaded objects in ACL2.* In *Practical Aspects of Declarative Languages*, LNCS 2257, pages 9–27, Springer–Verlag, 2002.
4. J. Caldwell. *Classical Propositional Decidability via Nuprl Proof Extraction* Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics, (TPHOLs'98). pg 105-122, LNCS 1479, Springer, 1998.
5. M. Davis and H. Putnam. *A computing procedure for quantification theory.* Journal of the Association for Computing Machinery, 7(3): 201–215, 1960.

6. M. Davis, G. Logemann and D. Loveland. *A machine program for theorem-proving.* Communications of the Association for Computing Machinery, 5(7): 394–397, 1962.
7. N. Dershowitz and Z. Manna. Proving Termination with Multiset Orderings. In *Proceedings of the Sixth International Colloquium on Automata, Languages and Programming*, LNCS 71, pages 188–202. Springer–Verlag, 1979.
8. M.C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer–Verlag, New York, 1996.
9. J. Gu, P. W. Purdom, J. Franco and B. W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In *Satisfiability Problem: Theory and Applications*, DIMACS: Series in Descrete and Applied Mathematics and Computer Science, vol. 35, American Mathematical Society, 1997.
10. J.H. Gallier. *Logic for Computer Science, Foundations of Automatic Theorem Proving.* Harper and Row Publishers, 1986.
11. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
12. M. Kaufmann and J S. Moore. *ACL2 Version 2.7*, 2001.
Homepage: `http://www.cs.utexas.edu/users/moore/acl2/`
13. M. Kaufmann and J S. Moore. *Structured Theory Development for a Mechanized Logic.* Journal of Automated Reasoning, 26(2): 161–203, 2001.
14. F.J. Martin–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory, 2002. 3rd Intl. Workshop on the ACL2 Theorem Prover and its Applications. Grenoble, 2002.
15. F.J. Martin–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. Verification in ACL2 of a generic framework to synthesize SAT-provers. In *Logic Based Program Synthesis and Tranformation*, LNCS 2664. Springer–Verlag, 2003.
16. F.J. Martin–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. *A generic framework for SAT-provers (formalization in ACL2).*
`http://www.cs.us.es/clg/theories/acl2/gen-sat`
17. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martin. Termination in ACL2 using multiset relation In *Thirty Five Years of Automating Mathematics*, Applied Logic Series, vol. 28, Kluwer Academic Publishers, 2003.
18. N. Shankar. *Little Engines of Proof.* In *FME 2002: Formal Methods - Getting IT Right*, LNCS 2391. Springer–Verlag, 2002.
19. R.M. Smullyan. *First-Order Logic.* Springer–Verlag: Heidelberg, Germany, 1968.
20. H. Zhang and M.E. Stickel. Implementing the Davis–Putnam method *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.