Verification in ACL2 of a Generic Framework to Synthesize SAT–Provers[⋆]

F.J. Martín–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina

Departamento de Ciencias de la Computación e Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
`http://www.cs.us.es/{~fmartin, ~jalonso,~mjoseh,~jruiz}`

Abstract. We present in this paper an application of the ACL2 system to reason about propositional satisfiability provers. For that purpose, we present a framework where we define a generic transformation based SAT–prover, and we show how this generic framework can be formalized in the ACL2 logic, making 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 automatized way. Three case studies are considered: semantic tableaux, sequent and Davis–Putnam methods.

1 Introduction

ACL2 [8] is a programming language, a logic for reasoning about programs in the language, and a theorem prover supporting formal reasoning in the logic. These components make ACL2 a particularly suitable system for reasoning about decision procedures, since proving and computing tasks can be done in the same system. Efficiency is one of the design goals of the system. Usually, it is obtained by building specific procedures to solve concrete problems. On the other hand, system characteristics make possible the development of generic procedures based on logic specifications. These generic procedures can be instantiated to obtain concrete ones [9], and this instantiation can be done maintaining efficiency to some extent.

In this paper, we describe an application of the ACL2 system to reason formally about a family of propositional satisfiability decision procedures. The common pattern of these 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. We will also describe how this instantiation process can be automatized. Three case studies are considered: semantic tableaux, sequent calculus and the Davis–Putnam method.

⋆ This work has been supported by project TIC2000-1368-C03-02 (Ministry of Science and Technology, Spain)

© Springer-Verlag Berlin Heidelberg 2003

This paper 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 method) can be placed into the generic framework. In section 3 we show how this framework can be formalized in ACL2 and how its main properties can be established. In section 4 we describe how these generic definitions and theorems can be instantiated in an automatized way, to obtain verified and executable Common Lisp definitions of tableaux based, sequent based and Davis–Putnam SAT–provers. Finally, in section 5 we draw some conclusions and discuss future work.

2 A Generic Framework to Develop Propositional SAT–Provers

Analyzing some well–known methods of proving propositional satisfiability (such as sequents, tableaux or Davis–Putnam), 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 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 can see this behavior in the semantic tableaux method with an example (figure 1–left). From the formula $(p \to q) \wedge p$ a tree 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. 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. The left branch becomes closed (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. In figure 1–right we can see how the sequent method behaves in a similar way, where objects are now sequents.

So our goal in this section is to describe a generic framework where these methods can be placed. First we introduce some notation. We consider an infinite set of 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 $\leftrightarrow$. $\overline{F}$ denotes the complementary of a formula F. 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

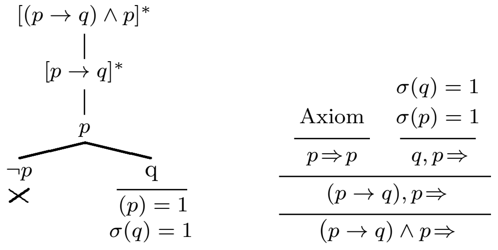

Fig. 1. An example of tableaux and sequents methods

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 sequences of formulas. We will use the notation $\langle e_1, ..., e_k \rangle$ to represent a finite sequence, and O^* to denote the set of sequences of elements from the set O. 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.

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

We will call $\mathcal{O}$ the set of *propositional objects* (or simply *objects*) and $\mathcal{R}$ the set of *expansion rules*. An element $(O, L) \in \mathcal{R}$ will be denoted as $O \rightsquigarrow_\mathcal{G} L$. Note that we allow rules of the form $O \rightsquigarrow_\mathcal{G} \langle\rangle$ and rules of the form $O \rightsquigarrow_\mathcal{G} \mathbf{t}$. When $(O, \sigma) \in \mathcal{V}$ we say that σ is a *distinguished valuation* for O, denoted as $\sigma \models_\mathcal{G} O$.

Definition 2. *Given a PTS $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$:*

1. *A* computation rule *is a function $r : \mathcal{O} \longrightarrow \mathcal{O}^* \cup \{\mathbf{t}\}$ such that $r \subseteq \mathcal{R}$.*
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 $\sigma : \mathcal{O}_\mathbf{t} \longrightarrow \mathbb{V}_\Sigma$, where $\mathcal{O}_\mathbf{t} = \{O \in \mathcal{O} : O \rightsquigarrow_\mathcal{G} \mathbf{t}\}$.*

Given a PTS $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V} \rangle$, 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 3 **($SAT_\mathcal{G}$)** *The input to this algorithm is a propositional formula F and it proceeds as follows:*

1. *Initially the list of objects $\langle i(F) \rangle$ is considered.*
2. *Given a list of objects $\langle O_1, ..., O_n \rangle$, an element O_j is selected.*
   (a) *If $r(O_j) = \mathbf{t}$, then the algorithm stops returning $\langle O_j \rangle$.*
   (b) *If $r(O_j) = \langle O'_1, ..., O'_m \rangle$, then the algorithm returns to point 2 with the list $\langle O'_1, ..., O'_m, O_1, ..., O_{j-1}, O_{j+1}, ..., O_n \rangle$.*
3. *If the list of objects becomes empty, the algorithm stops returning $\mathbf{f}$.*

The intuitive idea is simple: given F, we start with the initial object $i(F)$ and repeatedly apply the rules of $\mathcal{R}$ 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). In the first case, from an object O_j such that $r(O_j) = \mathbf{t}$ we can obtain a distinguished valuation using a model function, which turns out to be a model of the original formula. In the second case, the original formula is unsatisfiable. Let us now establish the main properties of this generic algorithm.

Definition 4. *We say that $SAT_{\mathcal{G}}$ is* complete *if for all $F \in \mathbb{P}(\Sigma)$ such that $\exists\sigma : \sigma \models F$, 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 $\exists\sigma : \sigma \models F$.*

Theorem 1. *Let $\mathcal{G} = \langle \mathcal{O}, \mathcal{R}, \mathcal{V}\rangle$ be a PTS, r a computation rule, i a representation function, μ a measure function and σ a model function, such that the following properties holds:*

$\mathcal{P}_1$*:* $O_i \in r(O) \Longrightarrow \mu(O_i) < \mu(O)$
$\mathcal{P}_2$*:* $F \in \mathbb{P}(\Sigma) \Longrightarrow (\sigma \models F \Longleftrightarrow \sigma \models_{\mathcal{G}} i(F))$
$\mathcal{P}_3$*:* $O \in \mathcal{O} \wedge r(O) \neq \mathbf{t} \Longrightarrow (\sigma \models_{\mathcal{G}} O \Longleftrightarrow \exists O_i \in r(O), \sigma \models_{\mathcal{G}} O_i)$
$\mathcal{P}_4$*:* $O \in \mathcal{O} \wedge r(O) = \mathbf{t} \Longrightarrow \sigma(O) \models_{\mathcal{G}} O$

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 $\sigma(O) \models F$.

Termination Proof: To prove termination of $SAT_{\mathcal{G}}$ 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 $r(O_j) = \langle O'_1, ..., O'_m\rangle$.

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 as proved in [5]). 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, as 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 $\mathcal{P}_4$, $\sigma(O) \models_{\mathcal{G}} O$. Then, by the property noted in the completeness proof, in every list considered in point 2 exists O such that $\sigma(O) \models_{\mathcal{G}} O$. Therefore, this holds for the initial list considered $\langle i(F)\rangle$, i.e., $\sigma(O) \models_{\mathcal{G}} i(F)$, and, by $\mathcal{P}_2$, $\sigma(O) \models F$. □

2.1 Semantic Tableaux

We consider the semantic tableaux method as it is described in [6]. The tableaux expansion rules are concisely presented using the uniform notation [14]. Using this notation, non–literal formulas are classified as doubly negated, α–formulas (equivalent to the conjunction of two components α_1 and α_2) or β–formulas (equivalent the a disjunction of two components, β_1 and β_2)[1].

We now describe the PTS $\mathcal{T} = \langle \mathcal{O}_\mathcal{T}, \mathcal{R}_\mathcal{T}, \mathcal{V}_\mathcal{T} \rangle$ associated with the semantic tableaux method. In this PTS, $\mathcal{O}_\mathcal{T}$ is the set of tableau branches (represented as lists of formulas), $\mathcal{V}_\mathcal{T}$ is the set of pairs (θ, σ) such that σ makes true every formula in θ, and $\mathcal{R}_\mathcal{T}$ the set of rules given by the following rule schemata:

$$\boxed{\begin{array}{l}
\mathcal{RT}_1 : \langle \Gamma_1, G, \Gamma_2, \neg G, \Gamma_3 \rangle \rightsquigarrow_\mathcal{T} \langle\rangle \\
\mathcal{RT}_2 : \langle \Gamma_1, \neg\neg G, \Gamma_2 \rangle \rightsquigarrow_\mathcal{T} \langle \langle \Gamma_1, G, \Gamma_2 \rangle \rangle \\
\mathcal{RT}_3 : \langle \Gamma_1, \alpha, \Gamma_2 \rangle \rightsquigarrow_\mathcal{T} \langle \langle \Gamma_1, \alpha_1, \alpha_2, \Gamma_2 \rangle \rangle \\
\mathcal{RT}_4 : \langle \Gamma_1, \beta, \Gamma_2 \rangle \rightsquigarrow_\mathcal{T} \langle \langle \Gamma_1, \beta_1, \Gamma_2 \rangle, \langle \Gamma_1, \beta_2, \Gamma_2 \rangle \rangle \\
\mathcal{RT}_5 : \Gamma \rightsquigarrow_\mathcal{T} \mathbf{t} \text{ if } \Gamma \text{ does not have non–literal} \\
\qquad\;\; \text{nor complementary formulas}
\end{array}}$$

We define the representation function $i_\mathcal{T}$ such that for every $F \in \mathbb{P}(\Sigma)$, $i_\mathcal{T}(F) = \langle F \rangle$. Obviously, $\sigma \models F \iff \sigma \models_\mathcal{T} i(F)$ (property $\mathcal{P}_2$).

We consider any computation rule, $r_\mathcal{T}$, such that, for every branch θ, $r_\mathcal{T}(\theta)$ is a rule from the above table that can be applied to θ.

We define the uniform measure, 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 $u(F) < u(\neg\neg F)$. 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 $\neg\neg F$ is also logically equivalent to F. 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 the definition of computation rule, this trivially implies property $\mathcal{P}_3$.

Finally, we define the model function $\sigma_\mathcal{T}$ such that for every branch θ without non–literal nor complementary formulas, $\sigma_\mathcal{T}(\theta) \models p$ if and only if p is a positive literal occurring in θ. Obviously if $r_\mathcal{T}(\theta) = \mathbf{t}$ then $\sigma_\mathcal{T}(\theta) \models_\mathcal{T} \theta$ (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–left performs the following steps (represented as $\longmapsto_{SAT_\mathcal{T}}$):

$$\begin{array}{l}\langle\langle (p \rightarrow q) \wedge p \rangle\rangle \longmapsto_{SAT_\mathcal{T}} \langle\langle p \rightarrow q, p \rangle\rangle \longmapsto_{SAT_\mathcal{T}} \\ \qquad\qquad \longmapsto_{SAT_\mathcal{T}} \langle\langle p, \neg p \rangle, \langle p, q \rangle\rangle \longmapsto_{SAT_\mathcal{T}} \langle\langle p, q \rangle\rangle \longmapsto_{SAT_\mathcal{T}} \langle\langle p, q \rangle\rangle\end{array}$$

2.2 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

[1] We extend the uniform notation to include equivalence: $F \leftrightarrow G$ is considered a β–formula with components $\beta_1 = F \wedge G$, $\beta_2 = \neg F \wedge \neg G$, and $\neg(F \leftrightarrow G)$ is considered a β–formula with components $\beta_1 = F \wedge \neg G$, $\beta_2 = \neg F \wedge G$.

sequent $\Gamma \Rightarrow \Delta$ true if and only if $\exists X \in \Gamma, (\sigma \not\models X) \vee \exists Y \in \Delta, (\sigma \models Y)$. We will consider the axioms and rules of Gentzen System G' presented in [7], with two additional rules about equivalence:

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

In order to prove the satisfiability of a formula F, the rules are applied to the initial sequent $F \Rightarrow$ until atomic sequents are obtained. Some atomic sequents provide countermodels of the initial sequent, and hence, models of the original formula. See [7] for more background about the sequent method.

We now describe the PTS $\mathcal{S} = \langle \mathcal{O}_\mathcal{S}, \mathcal{R}_\mathcal{S}, \mathcal{V}_\mathcal{S} \rangle$ associated with the sequents method. In this PTS, $\mathcal{O}_\mathcal{S}$ is the set of sequents (represented as pairs of lists of formulas), $\mathcal{V}_\mathcal{S}$ is the set of pairs (S, σ) such that σ makes S false and $\mathcal{R}_\mathcal{S}$ the set of rules given by the following rule schemata:

$$\boxed{\begin{array}{l}
\mathcal{R}s_1 : \langle \Gamma_1, F, \Gamma_2 \rangle \Rightarrow \langle \Delta_1, F, \Delta_2 \rangle \rightsquigarrow_\mathcal{S} \langle\rangle \\
\mathcal{R}s_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{R}s_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{R}s_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{R}s_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{R}s_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{R}s_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{R}s_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{R}s_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{R}s_{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{R}s_{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{R}s_{12} : \Gamma \Rightarrow \Delta \rightsquigarrow_\mathcal{S} \mathbf{t} \text{ if } \Gamma \Rightarrow \Delta \text{ is an atomic sequent and } \Gamma \cap \Delta = \emptyset
\end{array}}$$

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 a rule from the above table that can be applied to S.

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 the definition of computation rule, this implies property $\mathcal{P}_3$.

Finally, we define the model function $\sigma_\mathcal{S}$ such that for every atomic non–axiom sequent S, $\sigma_\mathcal{S}(S) \models p$ if and only if p occurs in the left part of S. Obviously, if $r_\mathcal{S}(S) = \mathbf{t}$ then $\sigma_\mathcal{S}(S) \models_\mathcal{S} S$ (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 1–right performs the following steps (represented as $\longmapsto_{SAT_\mathcal{S}}$):

$$\langle\langle (p \to q) \wedge p \rangle \Rightarrow\rangle \longmapsto_{SAT_{\mathcal{S}}} \langle\langle p \to q, p \rangle \Rightarrow\rangle \longmapsto_{SAT_{\mathcal{S}}} \langle p \Rightarrow p, \langle q, p \rangle \Rightarrow\rangle \longmapsto_{SAT_{\mathcal{S}}}$$
$$\longmapsto_{SAT_{\mathcal{S}}} \langle\langle q, p \rangle \Rightarrow\rangle \longmapsto_{SAT_{\mathcal{S}}} \langle\langle q, p \rangle \Rightarrow\rangle$$

2.3 Davis–Putnam Method

The Davis–Putnam method is a procedure to decide the satisfiability of a set of clauses. Then, if $\mathcal{FC}$ is a procedure to obtain a set of clauses logically equivalent to a formula F, we can use the Davis–Putnam method to decide the satisfiability of F, applying the method to $\mathcal{FC}(F)$. See [6] for more background about the Davis–Putnam method.

We now describe the PTS $\mathcal{D} = \langle \mathcal{O}_{\mathcal{D}}, \mathcal{R}_{\mathcal{D}}, \mathcal{V}_{\mathcal{D}} \rangle$ associated with the Davis–Putnam method. Now, $\mathcal{O}_{\mathcal{D}}$ is the set of pairs $\langle S, M \rangle$, where S is a set of clauses and M is a literal list without complementary literals and such that for every L in M, neither L nor $\overline{L}$ is in some clause in S. $\mathcal{V}_{\mathcal{D}}$ is the set of pairs $(\langle S, M \rangle, \sigma)$ such that σ is model of every clause in S and every literal in M. $\mathcal{R}_{\mathcal{D}}$ is the set of rules given by the following rule schemata:

$$\boxed{\begin{array}{l}
\mathcal{R}_{\mathcal{D}1} : \langle S, M \rangle \rightsquigarrow_{\mathcal{D}} \langle\rangle \text{ if the empty clause is in } S \\
\mathcal{R}_{\mathcal{D}2} : \langle S, \langle L_1, \dots, L_n \rangle \rangle \rightsquigarrow_{\mathcal{D}} \langle\langle S_L, \langle L, L_1, \dots, L_n \rangle\rangle\rangle \\
\qquad\quad \text{if the unitary clause } \{L\} \text{ is in } S \\
\mathcal{R}_{\mathcal{D}3} : \langle S, \langle L_1, \dots, L_n \rangle \rangle \rightsquigarrow_{\mathcal{D}} \langle\langle S_L, \langle L, L_1, \dots, L_n \rangle\rangle, \langle S_{\overline{L}}, \langle \overline{L}, L_1, \dots, L_n \rangle\rangle\rangle \\
\qquad\quad \text{where } L \text{ is a literal in a clause in } S \\
\mathcal{R}_{\mathcal{D}4} : \langle\langle\rangle, M \rangle \rightsquigarrow_{\mathcal{D}} \mathbf{t}
\end{array}}$$

where S is a set of clauses, $\langle S, M \rangle$ and $\langle S, \langle L_1, \dots, L_n \rangle \rangle$ are elements in $\mathcal{O}_{\mathcal{D}}$, and $S_L = \{C - \{\overline{L}\} : C \in S \text{ and } L \notin C\}$ and $S_{\overline{L}} = \{C - \{L\} : C \in S \text{ and } \overline{L} \notin C\}$.

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 one of the expansion rules schemata in the following preference order $\mathcal{R}_{\mathcal{D}1}$, $\mathcal{R}_{\mathcal{D}2}$, $\mathcal{R}_{\mathcal{D}3}$ and $\mathcal{R}_{\mathcal{D}4}$.

We define the measure function $\mu_{\mathcal{D}}$, such that, for every pair $\langle S, M \rangle \in \mathcal{O}_{\mathcal{D}}$, $\mu_{\mathcal{D}}(\langle S, M \rangle)$ 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 $\langle S, M \rangle \rightsquigarrow_{\mathcal{D}} L$ 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 the definition of computation rule, this implies property $\mathcal{P}_3$.

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

$$\langle\langle\langle \{\neg p, q\}, \{p\} \rangle, \langle\rangle \rangle\rangle \longmapsto_{SAT_{\mathcal{D}}} \langle\langle\langle \{q\} \rangle, \langle p \rangle \rangle\rangle \longmapsto_{SAT_{\mathcal{D}}}$$
$$\longmapsto_{SAT_{\mathcal{D}}} \langle\langle\langle\rangle, \langle q, p \rangle \rangle\rangle \longmapsto_{SAT_{\mathcal{D}}} \langle\langle\langle\rangle, \langle q, p \rangle \rangle\rangle$$

3 Formalizing the Generic SAT–Prover in ACL2

Let us see in this section how we formalize the concepts and results of the previous section in the ACL2 logic.The ACL2 logic is a quantifier–free, first–order logic with equality, describing an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp. The logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference include those for propositional calculus, equality, and instantiation, as well as the introduction of new total recursive functions by the *principle of definition* (using `defun`) and constrained functions (via `encapsulate`). The ACL2 theorem prover mechanizes that logic, being particularly well suited for obtaining automatized proofs based on simplification and induction. For a detailed description of ACL2, we refer the reader to the ACL2 book [8].

3.1 Definition of the Generic Algorithm

Before reasoning in ACL2 about the algorithm $SAT_{\mathcal{G}}$, we have to define in the ACL2 logic the functions introduced by the generic framework presented in section 2. 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)	$\sigma(O)$
`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, 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. In this case, the properties about the generic functions are the following[2]:

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

Theorem: `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)

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

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

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

Theorem: `P2`
`propositional-p`(F)
  $\to$ (`gen-dist-val`(σ, `gen-repr`(F)) $\leftrightarrow$ `models`(σ, F))

Definition:

```
gen-dist-val-lst(σ, O-lst) =
if endp(O-lst) then nil
else gen-dist-val(σ, car(O-lst))
       ∨ gen-dist-val-lst(σ, cdr(O-lst))
fi
```

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

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

Theorem: `gen-select-member`
`consp`(lst) $\to$ (`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 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 (abstractly) defined, we define in ACL2 the function `generic-sat`, implementing the algorithm $SAT_{\mathcal{G}}$:

[3] 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-lst` can be seen as a generalized disjunction of the predicate `gen-dist-val` acting on the objects of a list.

```
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)
     fi
fi
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. When a rule of the form $\langle O, \mathtt{t} \rangle$ 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 will be 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).

3.2 Termination

By the ACL2 principle of definition, new function definitions are admitted as axioms only if there exists a well–founded measure in which the arguments of each recursive call decrease, ensuring in this way that no inconsistencies are introduced by new definitions. In the case of the function `generic-sat-lst` the heuristics of the prover are not able to find a suitable termination argument, so we must explicitly provide a measure on its argument that decreases in every recursive call with respect to a well–founded relation.

In ACL2, the only primitive well–founded relation is `e0-ord-<`, the "less than" relation between ordinals up to ε_0, represented in terms of lists and natural numbers, given by the predicate `e0-ordinalp`. Nevertheless, the user can introduce a new well founded relation by providing the corresponding monotone ordinal function.

To show termination of `generic-sat-lst`, we follow the lines described in the informal proof given in section 2. 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 well founded relation `mul-e0-ord-<` (denoted as $<_{mul}$), defined as the multiset relation induced by `e0-ord-<`. Since `e0-ord-<` is well–founded, so is its induced multiset relation [5]. This result was formalized in the ACL2 logic in [12], where a macro named `defmul` was also developed. This macro automatically generates the definitions and theorems needed to define a well–founded multiset relation induced by a given well–founded relation.

The main termination property of `generic-sat-lst` is given by the following theorem, establishing that this measure decreases in every recursive call, and allowing the admission of the function `generic-sat-lst`.

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

3.3 Soundness and Completeness

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

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

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

These two theorems formalize theorem 1 in ACL2. They are proved along the lines of the informal proof given in section 2, basically first proving by induction analogue properties about the function `generic-sat-lst`. Here the properties assumed about the generic functions showed in the subsection 3.1 plays a crucial role. The function `generic-mod`, provides a model of a satisfiable formula, its definition is the following:

DEFINITION:
$\texttt{generic-mod}(F) = \texttt{gen-model}(\texttt{first}(\texttt{generic-sat}(F)))$

4 Instantiating the Generic Framework

Concrete SAT–prover will be given by defining concrete counterparts of the abstractly defined functions given in subsection 3.1. With these concrete functions, one can define concrete version of the algorithm `generic-sat`.

A derived rule of inference in ACL2, *functional instantiation*, allows some kind of second–order reasoning: theorems about previously defined (or abstractly defined) functions can be instantiated with function symbols known to have the same properties. In this case, 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 A Tableaux Based SAT–Prover

Along the lines of subsection 2.1, we can define in ACL2 a tableaux based instantiation of the generic framework. For that purpose we define a tableaux version of the generic functions given in subsection 3.1: `tableaux-object-p`, `tableaux-repr`, `tableaux-comp-rule`, `tableaux-dist-val`, `tableaux-model`, `tableaux-measure` and `tableaux-select`. These functions are defined as suggested in subsection 2.1. In this case, objects are lists of propositional formulas, representing branches in a tableau.

For example, the definition of the computation rule is the following:

DEFINITION:
```
tableaux-comp-rule($\theta$) =
if closed-tableau($\theta$) then nil                                  $\mathcal{RT}_1$
else let $F$ be one-formula($\theta$)
        in
    if doubly-neg-p($F$)
    then list(add(neg-neg-component($F$), remove-one($F$, $\theta$)))   $\mathcal{RT}_2$
    elseif alfa-formula-p($F$)
    then list(add(component-1($F$),
                  add(component-2($F$), remove-one($F$, $\theta$))))    $\mathcal{RT}_3$
    elseif beta-formula-p($F$)
    then list(add(component-1($F$), remove-one($F$, $\theta$)),
              add(component-2($F$), remove-one($F$, $\theta$)))         $\mathcal{RT}_4$
    else t                                                              $\mathcal{RT}_5$
    fi
fi
```

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

Note that this computation rule implements a strategy for applying the tableaux expansion rules in a preference order, given by a 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 elaborated.

Once the assumed properties in the generic framework have been proved for the tableaux case, we can instantiate the generic SAT–prover algorithm, and prove analogue theorems of termination, soundness and completeness, but now using functional instantiation. 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 [9], we describe a user tool 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.

We defined a macro named `make-generic-theory`, which receives as argument a list of ACL2 events (definitions and theorems) that can be instantiated. When an ACL2 book[4] developing a generic theory is created, we include a call to this macro in its last line, for example, in the book that formalizes the generic framework for SAT–provers (as described in the previous section), we include the following last call:

```
(make-generic-theory *generic-sat*)
```

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

For example, once defined the functions implementing the tableaux counterparts of the generic functions, when we include the book with the generic SAT–prover formalization, a macro `definstance-*generic-sat*` is automatically defined, and we can use this macro to automatically generate instantiated events for the tableux based SAT–prover, as follows:

```
(definstance-*generic-sat*
  ((gen-object-p        tableaux-object-p)
   (gen-repr            tableaux-repr)
   (gen-dist-val        tableaux-dist-val)
   (gen-dist-val-list   tableaux-dist-val-list)
   (gen-comp-rule       tableaux-comp-rule)
   (gen-select          tableaux-select)
   (gen-measure         tableaux-measure)
   (gen-model           tableaux-model))
   "-tableaux")
```

Note that this macro receives as input a functional substitution, associating every function of the generic framework with its tableaux counterpart. It also

[4] 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.

receives a string, used to name the new events generated, by appending it to the name of the original event.

The result of this macro call is the *automatic* generation of the events needed to define and verify in ACL2 a tableaux based propositional SAT–prover. As a consequence, the definition of a function named `generic-sat-tableaux` is generated in an analogue way to `generic-sat` (using the tableaux auxiliary functions). And also the following theorems, establishing the soundness and completeness of `generic-sat-tableaux` are automatically generated and proved:

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

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

Note that, once proved that the tableaux counterparts of the generic functions verify the properties showed in subsection 3.1, no additional proof effort is needed to define and verify the tableaux–based SAT–prover.

4.2 Sequent and Davis–Putnam Based SAT–Provers

We can follow an analogous procedure to define and verify, sequent and Davis–Putnam 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 2.2, for the sequent based SAT–prover, and 2.3, for the Davis–Putnam based SAT–prover.

4.3 Execution Examples

The functions implementing the previous SAT–provers are executable in any compliant Common Lisp (with the appropriated files loaded). In the following table we present the results of applying the tableaux and sequents procedures to prove the satisfiability of a propositional version of the N-queens problem[5]. We also apply the Davis–Putnam procedure to the same problem. Note that the Davis–Putnam procedure works with propositional clauses and a previous translation of propositional formulas into clauses is needed in this case (we do not include the translation times).

[5] All results are in seconds of user CPU on a double 800MHz Pentium III.

N	Tableaux	Sequents	Davis-Putnam
2	0.010	0.000	0.000
3	0.060	0.020	0.010
4	0.530	0.180	0.040
5	2.370	0.820	0.140
6	212.070	72.600	0.250
7	750.540	255.640	0.570

The complete files with definitions and theorems about the generic framework, the instances and the examples, are available on the Web in `http://www.cs.us.es/~fmartin/acl2-gen-sat/`

5 Conclusions and Further Work

We have presented an application of the ACL2 theorem prover to reason about SAT decision procedures. First, we considered a generic SAT–prover, having the essential properties of every transformation based SAT–prover. Second, we reasoned about the generic algorithm, establishing its main properties. And third, we obtained verified and executable SAT–provers using functional instantiation. This last process can be done in an automatized way.

The main effort in the formalization of the generic SAT–prover has been done in the termination proof of the generic algorithm. This proof is based on a previous work about multiset relations [12]. With respect to the instantiation procedure, the main effort has been done in the proof of the concrete version of properties `P3` and `P4` and the development of the instantiation tool. A detailed presentation of this tool can be found in [9]. The following table summarizes the number of definitions, theorems and hints needed to formalize and prove each section:

Section	Definitions	Theorems	Hints
Generic algorithm	15	37	13
Tableaux based SAT–prover	23	53	13
Sequent based SAT–prover	21	37	9
Davis–Putnam SAT–prover	23	47	9

There is some related work in mechanical verification of SAT–provers. A classical example is Boyer and Moore's propositional tautology checker [3], 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. See this reference for an additional account of related works.

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.

We have used this methodology to develop resolution based SAT–provers: we have defined a generic resolution procedure and we have proved its termination, soundness and completeness properties. The completeness proof is based on the developed by Bezem in [2]. This work can be found in [10].

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 this line of work we have implemented a DPLL (Davis Putnam Logemann Loveland) procedure in ACL2, based on [15], which turns out to be much more efficient than the one presented here (It solves the 7-queens problem in 0.010 seconds and the 16-queens problem in 0.750 seconds). Using this technique of refinement, we could verify this more efficient version of the Davis–Putnam procedure

Another line of work is to apply the generic framework to other SAT methods (KE, TAS-D [1], etc). Finally, we also plan to use the same methodology to develop generic framework for non–classical and first–order logics. In the last case, we think that the development of a generic framework could be easy, combining the results presented here with a verified unification algorithm, such as the algorithm presented in [13] and [11].

References

1. G. Aguilera, I.P. de Guzman, M. Ojeda–Aciego and A. Valverde. *Reductions for non-clausal theorem proving.* Theoretical Computer Science 266, pages 81–112. Elsevier, 2001.
2. M. Bezem. *Completeness of resolution revisited.* Theoretical Computer Science 74, no. 2, pages 227–237, 1990.
3. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, 1979.
4. J. Caldwell. *Decidability Extracted: Synthesizing "Correct-by-Construction" Decision Procedures from Constuctive Proofs.* PhD thesis, Cornell University, 1998
5. 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.
6. M.C. Fitting. *First–Order Logic and Automated Theorem Proving.* Springer–Verlag, New York, 1990.
7. J.H. Gallier. *Logic for Computer Science, Foundations of Automatic Theorem Proving.* Harper and Row Publishers, 1986.
8. M. Kaufmann, P. Manolios, and J S. Moore. *Computer–Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
9. 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.
10. F.J. Martin–Mateos. Teoria computacional (en ACL2) sobre calculos proposicionales. PhD thesis, University of Seville, 2002.
11. J.L. Ruiz–Reina. Una teoria computacional acerca de la logica ecuacional. PhD thesis, University of Seville, 2001.

12. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martin. Multiset Relations: a Tool for Proving Termination. In *Second ACL2 Workshop*, Technical Report TR-00-29, Computer Science Departament, University of Texas, 2000.
13. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martin. Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover. In *Proceedings AGP'99, Joint Conference on Declarative Programming*, L'Aquila (Italia), 1999.
14. R.M. Smullyan. *First–Order Logic.* Springer–Verlag: Heidelberg, Germany, 1968.
15. H. Zhang and M.E. Stickel. Implementing the Davis–Putnam method *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.