Formal verification of propositional SAT provers*

F.J. Martín–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina
http://www.cs.us.es/{~fmartin,~jalonso,~mjoseh,~jruiz}

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

Abstract. We present in this paper a formal approach to the verification of a family of propositional SAT provers. For that purpose we use the ACL2 system, a theorem prover for reasoning about programs written in an applicative subset of Common Lisp. We developed 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 automated way. In particular, this approach is applied to the formal verification of Common Lisp implementations of propositional provers based on tableaux, sequents and the Davis–Putnam procedure, respectively.

1 Introduction

Determining whether a propositional theory is satisfiable (known as the *SAT problem*) is very important in many fields of Artificial Intelligence: many problems that occur in planning, knowledge representation, learning, and other areas of AI are essentially satisfiability problems [2]. Thus, formal verification of algorithms solving SAT is interesting as a way to provide a *certification* of the correctness of the computations performed.

In this paper, we describe the application of the ACL2 system [3] to reason about a family of propositional SAT provers. ACL2 is both a programming language and a logic that allows formal reasoning about programs in that language. It is also a theorem prover giving mechanized support for proving theorems in the logic. Thus, it provides a single framework where both proving and computing are possible.

One of the main features of our approach is that the formal verification is carried out with a high level of abstraction. For that purpose, we define a generic transformation based SAT–prover, a common abstract pattern for a family of well–known SAT algorithms. 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 then be instantiated to obtain a

* This work has been supported by DGES/MEC: Projects TIC2000-1368-C03-02 and PB96-1345

number of verified and executable SAT–provers in ACL2, and this can be done in an automated way. In particular, we have applied this technique to automatically obtain the formal verification of Common Lisp implementations for three well–known satisfiability algorithms based on tableaux, sequents and the Davis–Putnam procedure, respectively.

Due to the lack of space we will skip details of the mechanical proofs. The interested reader is urged to obtain the complete files with definitions and theorems, available on the web in `http://www.cs.us.es/~fmartin/acl2-gen-sat`, where also an extended version of this paper can be found.

2 A generic framework to develop propositional SAT–provers

We will assume that the reader has familiarity with the basic concepts and results of propositional logic (see [1], for example). 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*. The set of propositional formulas on Σ is denoted as $\mathbb{P}(\Sigma)$, where the basic connectives are $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. A valuation is a function $\sigma : \Sigma \longrightarrow \mathbb{B}$. 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). The *propositional satisfiability problem* can formulated as follows: given a propositional formula F, check whether there exists a model σ of F. A *SAT prover* is a program that finds such σ, whenever it exists.

In this section, we describe a generic framework where a family of well–known propositional SAT provers can be fit. The main idea is that we can consider all these methods as the iterative application of a set of *transformation rules* that are applied to some kind of *objects* built from formulas. To illustrate this, consider the example in Figure 1–left where the semantic tableaux method is applied to find a model of the formula $(p \rightarrow q) \wedge p$. Initially, a tree with a single node is built. In a first step the formula is expanded obtaining one extension with two formulas $p \rightarrow q$ and p. In a second step the formula $p \rightarrow q$ 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.

Representing a branch in the tree as the sequence of its formulas, and a tree as the sequence of its branches, we can alternatively describe this example by a sequence transformation rules that are applied to the branches of the tree (we use the notation $\langle e_1, ..., e_k \rangle$ to represent finite sequences):

$$\langle\langle (p \rightarrow q) \wedge p \rangle\rangle \longmapsto \langle\langle p \rightarrow q, p \rangle\rangle \longmapsto$$
$$\longmapsto \langle\langle p, \neg p \rangle, \langle p, q \rangle\rangle \longmapsto \langle\langle p, q \rangle\rangle \longmapsto \langle\langle p, q \rangle\rangle$$

Note that in every step, a branch is selected and a transformation rule is applied to it, obtaining a list of new branches that replace the selected branch. As a particular case, if a closed branch is selected, the empty list of branches is obtained, having the effect of deleting the selected branch. The transformation steps are applied until the simple branch $\langle p, q \rangle$ is selected. From this kind of

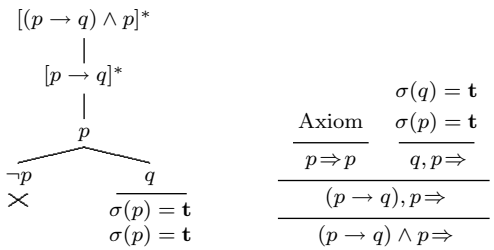

Fig. 1. An example of tableaux and sequents methods

simple branches (non–closed and without complementary literals) we can easily obtain a model, that turns out to be a model of the initial formula.

Other propositional methods can be seen in the same way. For example, in figure 1–right we can see how the sequent method behaves in a similar way, where objects are now sequents instead of branches of a tree. In fact, analyzing some well–known methods of proving propositional satisfiability (such as sequents, tableaux or Davis–Putnam), we can observe this common behavior. They do not work directly on formulas but on objects built from formulas. The objects are repeatedly modified using transformation rules reducing their complexity in such a way that their meaning is preserved. Eventually, from some kind of simple objects, one can obtain a valuation proving satisfiability of the original formula (which we will call a distinguished valuation). If no such object is found, then unsatisfiability of the original formula is proved. Based on this intuitive idea, we describe in the following subsection a formal description of a generic SAT–prover in ACL2.

2.1 Formalization in the ACL2 logic

ACL2 [3] stands for A Computational Logic for Applicative Common Lisp. ACL2 is a system that comprises a programming language (an applicative subset of Common Lisp), a logic that allows to formulate and prove properties about programs written in the language, and a theorem prover supporting mechanized reasoning in the logic. The ACL2 logic is a subset of the first-order logic with equality, without quantifiers. The syntax of its formulas is that of Common Lisp [7] (we will assume the reader familiar with this language). The logic includes axioms for propositional logic and for a number of standard Common Lisp functions and data types, describing their behavior. The propositional connectives are implemented by the functions `not`, `and`, `or`, `implies` and `iff`. Rules of inference includes those for propositional calculus, equality, and instantiation. By the *principle of definition*, new function definitions can be introduced in the logic. Thus, defining a function in ACL2 (by means of `defun`) has a double effect. First, a program is defined as usual in Common Lisp; second, the definition of the function is introduced as an axiom in the logic and then formal mechanized reasoning about it is possible.

Let us now deal with the ACL2 formalization of the generic SAT prover sketched at the beginning of this section. Recall that we can view this generic SAT prover as the iterative application of transformation rules to some kind of objects built from formulas. To obtain a formal definition of this process, assume for the moment that we have defined two functions `gen-repr` and `gen-comp--rule`. We will explain below how we introduce these functions in the logic. The intuitive idea is that `(gen-comp-rule obj)` is the *list of objects* obtained by applying one step of transformation rule to the object `obj`, and that `(gen-repr F)` builds the initial object from an initial input formula `F`. With this auxiliary functions, we can define the following function `generic-sat` that defines our generic SAT prover:

```
(defun generic-sat-lst (obj-lst)
  (if (endp obj-lst)
      nil
      (let* ((obj (gen-select obj-lst))
             (rest (remove-one (gen-select obj-lst) obj-lst))
             (expansion (gen-comp-rule obj)))
        (cond ((equal expansion t)
               (list obj))
              (t (generic-sat-lst (append expansion rest)))))))

(defun generic-sat (F)
  (generic-sat-lst (list (gen-repr F))))
```

The main function of this algorithm is given by the recursive function `generic-sat-lst`, acting on a list of objects to be expanded. In every step, an object from the list is selected and a transformation rule is applied, obtaining a new list of objects that is appended to the remaining list of objects to be expanded (the selection of an object of the list is implemented by a function `gen-select`). Note that there are some objects such that the application of a transformation step to them returns `t`, called *simple* objects. This is a way to recognize those objects that provide a valuation proving satisfiability of the original formula. The transformation rules are applied until there are no more objects to be expanded or until a simple object is selected. As we will see, the first case indicates unsatisfiability of the input formula and in the second case we can obtain a model of the input formula.

The auxiliary functions used by `generic-sat` (i.e. `gen-comp-rule`, `gen-repr` and `gen-select`) are not introduced in the ACL2 logic using the principle of definition. Since the algorithm described is an abstract pattern for different classes of SAT provers, the auxiliary functions are not defined completely. Instead, we only assume that they have certain "minimal" properties that ensure soundness and completeness of the algorithm described by `generic-sat`. This can be done in ACL2 by means of the `encapsulation` mechanism that 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).

Thus, we used encapsulation to assume the following properties about the auxiliary functions used (see the web page for a detailed description):

- In every transformation step, satisfiability of the set of involved formulas is preserved.
- When applying a transformation step to an object, every member of the list of objects obtained is smaller with respect to some well–founded measure.
- There exists a function `gen-model` such that when given as input a simple object it returns a model of the object.

2.2 Formal properties of the generic SAT prover

The following are the main theorems we proved in the ACL2 theorem prover, about the function `generic-sat`:

```
(defthm soundness-generic-sat
  (implies (and (propositional-p F) (generic-sat F))
           (models (generic-mod F) F)))

(defthm completeness-generic-sat
  (implies (and (propositional-p F) (models sigma F))
           (generic-sat F)))
```

The above theorems establish that `(generic-sat F)` is not `nil` if and only if there exists a model of `F` (in that case, the model of `F` is returned by the function `generic-mod`[1]). That is, the theorems formally establish the soundness and completeness of the generic procedure described by `generic-sat`.

The functions `propositional-p` and `models` used in the statement of these theorems, formalize the notions of propositional formula and model of a formula, respectively. They are part of a formalization we carried out in ACL2 of the syntax and semantic of propositional logic.

The above properties are proved by means of a typical interaction with the ACL2 theorem prover. It must be noted that the theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, in a deeper sense, the system is interactive. Very often, non-trivial proofs are not found by the system in the first attempt. We had to guide the prover by adding lemmas and definitions, used in subsequent proofs as rules, thus helping the prover to find a preconceived proof by means of a suitable set of rules. It is also remarkable that a considerable part of the total proof effort was invested to prove termination of the function `generic-sat-lst`, which is not trivial. See the web page for details.

[1] The function `generic-mod` is defined as `gen-model` acting on the single object returned by `(generic-mod F)`.

3 Instantiating the generic framework

In the previous section, we have described the definition and formal verification of a generic SAT–prover in ACL2. This generic prover is defined in terms of some auxiliary functions partially defined, assuming about them the minimal properties needed to prove soundness and completeness. Therefore the generic SAT–prover is not executable, but it can be seen as a pattern for some concrete and Common Lisp executable SAT–provers based on transformation rules.

The functions partially defined by encapsulation can be seen as second order variables, representing functions with those properties. A derived rule of inference in ACL2, *functional instantiation*, allows some kind of second–order reasoning: theorems about (partially) 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, having as a result an executable and formally verified Common Lisp implementation. This process can be automatized to some extent, as we will see. In the following we illustrate this method describing the definition and verification of a tableaux based SAT–prover.

To obtain a Common Lisp definition of the propositional tableaux method, together with its formal verification, we first define concrete executable versions of the generic auxiliary functions used in the definition and verification of the generic prover. For example, in this case we define (using `defun`) the functions `tableaux-repr`, `tableaux-comp-rule`, and `tableaux-select`, concrete implementations for the tableaux method, corresponding to the generic functions `gen--repr`, `gen-comp-rule`, and `gen-select`, respectively. The only requirement for these definitions is that the assumed properties in the generic case have to be met by the concrete functions in the tableaux case.

For example, the definition of the function `tableaux-comp-rule`, which implements the computation rule is the following:

```
(defun tableaux-comp-rule (branch)
  (if (closed-tableau branch)
      nil
      (let ((F (one-formula branch)))
        (cond ((doubly-neg-p F)
               (list (add (neg-neg-component F) (remove-one F branch))))
              ((alfa-formula-p F)
               (list (add (component-1 F)
                          (add (component-2 F) (remove-one F branch)))))
              ((beta-formula-p F)
               (list (add (component-1 F) (remove-one F branch))
                     (add (component-2 F) (remove-one F branch))))
              (t t)))))
```

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 whether the formula is of type α or of type β (see [1] for a precise description of these concepts and the tableaux method). 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 for `gen-comp-rule` can be proved for this concrete counterpart.

Once the assumed properties in the generic framework has 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 has 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 [4], we described 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 now briefly describe this generic instantiation tool (see [4] for a more detailed description). We defined a macro named `make-generic-theory`, which receives as argument a list of ACL2 events (definitions and theorems) that can be functionally instantiated. When an ACL2 book[2] 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-*sat-generico*
  ((gen-repr           tableaux-repr)
   (gen-comp-rule      tableaux-comp-rule)
   (gen-select         tableaux-select)
   ...)
   "-tableaux")
```

[2] 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.

Note that this macro receives as input a functional substitution, relating every function of the generic framework with its tableaux counterpart. It also 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 that define and verify in ACL2 a tableaux based propositional SAT–prover. That is, 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:

```
(defthm soundness-generic-sat-tableaux
  (implies (and (propositional-p F) (generic-sat-tableaux F))
           (models (generic-mod-tableaux F) F)))

(defthm completeness-generic-sat-tableaux
  (implies (and (propositional-p F) (models sigma F))
           (generic-sat-tableaux F)))
```

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

We have applied the same procedure as described above to the definition and verification of other propositional SAT–provers that can be fit in this transformation based paradigm. Namely, procedures based on the sequent calculus and on the Davis–Putnam procedure (see [1] for a description of these procedures). In both cases, we only have to define concrete versions for the auxiliary functions and prove that these concrete implementations satisfy the properties assumed in the generic case. Then we simply use our generic instantiation tool to automatically obtain executable Common Lisp implementations of these methods, with their corresponding theorems of soundness and completeness.

4 Execution examples

As we said above, the concrete verified definitions, generated by our generic instantiation tool from the generic SAT–prover, are executable in any compliant Common Lisp and (in particular) in the ACL2 system. Recall that these functions are implementations of a tableaux–based, sequent–based and Davis–Putnam procedures for proving propositional satisfiability. In this section we show some quantitative information about the execution of these SAT–provers[3]

In Figure 2 we show the use of the tableaux and sequents procedures to prove the validity of the formulas of Urquhart [9] for different values of the parameter N, by the unsatisfiability proof of its negation. In figure 3 we also present the results of applying the tableaux and sequents procedures to prove the

[3] The executions are carried out in double Pentium III - 800 MHz

satisfiability of a propositional version of the N-queens problem (see [5], chapter 13). 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 example (we do not include the translation times).

N	Tableaux	Sequents	N	Tableaux	Sequents	N	Tableaux	Sequents
1	0.000	0.000	6	0.060	0.000	11	6.150	0.660
2	0.000	0.000	7	0.210	0.020	12	14.230	1.470
3	0.010	0.000	8	0.440	0.040	13	32.580	3.450
4	0.000	0.000	9	1.120	0.130	14	73.860	7.760
5	0.030	0.010	10	2.610	0.270	15	166.380	17.480

Fig. 2. Times for Urquhart's formulas

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

Fig. 3. Times for N-queens problem

Although the execution times of these implementations are acceptable, the efficiency can be improved. In fact, we implemented the DPLL procedure in ACL2, based on [8], which turns out to be much more efficient (Figure 4). This procedure, although implemented in ACL2, is not formally verified. But having its definition in ACL2, this efficient procedure could be formally verified, and it is our intention to do so. The formal verification of this efficient versions can be done using the basic versions already verified, by proving equivalence theorems between both versions, a methodology that we already applied in other problems [6].

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 (namely tableaux, sequent and Davis–Putnam procedures) using functional instantiation. This last process can be done in a somewhat automated way.

Applying formal methods to the verification of SAT–provers is an interesting way to certify the computations performed by different implementations of a

Problem (Sat/Unsat)	DPLL (Total time)
aim-50 *(16/8)*	0.290
aim-100 *(16/8)*	69.710
hole *(0,5)*	1844.090
par-8 *(10,0)*	0.309
par-16 *(10,0)*	2281.950
jnh *(16,34)*	51.580
ii-8 *(14,0)*	107.340

Fig. 4. Times for some DIMACS problems

very important problem in Artificial Intelligence. We used the ACL2 system for this task because it provides a system where computation and formal proofs can be intermixed.

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.

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. This is a line of future work. Finally, we also plan to use the same methodology to develop a generic framework for resolution based theorem provers.

References

1. M.C. Fitting. *First–Order Logic and Automated Theorem Proving.* Springer–Verlag, New York, 1990.
2. *Journal of Automated Reasoning.* Special issue on "Satisfiability in the Year 2000", 24(1–2), 2000 and 28(2), 2002.
3. M. Kaufmann, P. Manolios, and J S. Moore. *Computer–Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
4. F.J. Martín–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. *A Generic Instantiation Tool and a Case Study: A Generic Multiset Theory.* Third International Workshop on the ACL2 Theorem Prover and Its Applications, Grenoble, 2002
5. N.J. Nilsson. *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann Publishers, 1998.
6. J.L. Ruiz, J.A. Alonso, M.J. Hidalgo and F.J. Martín Progress Report: Term Dags Using Stobjs Third International Workshop on the ACL2 Theorem Prover and Its Applications, Grenoble, 2002
7. G.L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.
8. Zhang, H. and M.E. Stickel Implementing the Davis–Putnam method *Journal of Automated Reasoning*, 24(1–2):277–296, 2000.
9. A. Urquhart Hard examples for resolution *Journal of the ACM*, 34(1):209–219, 2000.