Automated Certification of Implicit Induction Proofs

Sorin Stratulat, Vincent Demange

Automated Certification of Implicit Induction Proofs

Sorin Stratulat Vincent Demange

LITA, Paul Verlaine-Metz University, Ile du Saulcy, 57000, Metz, France
{stratulat,demange}@univ-metz.fr

Abstract. Theorem proving is crucial for the formal validation of properties about user specifications. With the help of the Coq proof assistant, we show how to certify properties about conditional specifications that are proved using automated proof techniques like those employed by the Spike prover, a rewrite-based implicit induction proof system. The certification methodology is based on a new representation of the implicit induction proofs for which the underlying induction principle is an instance of Noetherian induction governed by an induction ordering over equalities. We propose improvements of the certification process and show that the certification time is reasonable even for industrial-size applications. As a case study, we automatically prove and certify more than 40% of the lemmas needed for the validation of a conformance algorithm for the ABR protocol.

1 Introduction

Theorem proving is a crucial domain for validating properties about user specifications. The properties formally proved with the help of theorem provers are valid if the proofs are sound. Generally speaking, there are two methods to certify (the soundness of the) proofs: either i) by certifying the implementation of the inference systems; in this way, any generated proof is implicitly sound, or ii) by explicitly checking the soundness of the proofs generated by not-yet certified theorem provers using certified proof environments like Coq [25].

We are interested in certifying properties about conditional specifications using automated proof techniques like those employed by Spike [5,19,3], a rewrite-based implicit induction proof system. The implementation of Spike's inference system is spread over thousands of lines of OCaml [12] code. Its certification, as suggested by method i), would require a tremendous proving effort. For example, [11] reports a cost of 20 person year for the certification of the implementation of another critical software: an OS-kernel comprising about 8,700 lines of C and 600 lines of assembler. For this reason, we followed the method ii), that has been firstly tested manually in [24], then automatically on toy examples in [23]. The method directly translates every step of a Spike proof into Coq scripts, which distinguishes it from previous methods based on proof reconstruction techniques [7,10,13] that mainly transform implicit into explicit induction proofs.

In this paper, we report improvements in order to certify implicit induction proofs concerning industrial-size applications. The case study of our choice is the validation proof of a conformance algorithm for the ABR protocol [14]. An interactive proof

using PVS [18] was firstly presented in [16], then it has been shown in [17] that more than a third of the user interactions can be avoided using implicit induction techniques, Spike succeeding to prove 60% of the user-provided lemmas automatically. Now, a simpler but more restrictive version of the Spike inference system has been shown powerful enough to prove 2/3 out of these lemmas. Moreover, any generated proof has been automatically translated into a Coq script, then automatically certified by Coq. We stress the importance of the automatic feature since the proof scripts are in many cases big and hard to manipulate by the users. The bottom-line is that these improvements allowed us to certify big proof scripts in a reasonable time, 20 times faster than in [23].[1]

The structure of the paper is as follows: after introducing the basic notions and notations in Section 2, we present in Section 3 the restricted inference system and a new representation of the implicit induction proofs for which the underlying induction principle is an instance of Noetherian induction governed by an induction ordering over equalities. The conformance algorithm and its Spike specification are discussed in Section 4. In Section 5, we describe a full implicit induction proof of one of the lemmas used in the ABR proof, then explain in Section 6 its translation into Coq script following the new representation of the implicit induction proofs. We detail the improvements we have made to speed-up the certification process and give statistics about the certification of the proofs of any of the 33 lemmas proved with the restricted inference system. Further improvements are discussed at the end of the section, including the parallelisation of the certification process. The conclusions and directions for future work are given in the last section.

2 Background and Notations

This section briefly introduces the basic notions and notations related to proving properties about conditional specifications by implicit induction techniques. More detailed presentations of them, and about equality reasoning in general, can be found elsewhere, for example in [2].

We assume that $\mathcal{F}$ is an alphabet of arity-fixed function symbols and $\mathcal{V}$ is a set of universally quantified variables. The set of functions symbols is split into *defined* and *constructor* function symbols. We also assume that the function symbols and variables are *sorted* and that for each sort s there is at least one constructor symbol of sort s. The set of *terms* is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and the set of variable-free (ground) terms by $\mathcal{T}(\mathcal{F})$. The sort of a non-variable term of the form $f(\ldots)$ is the sort of f, where $f \in \mathcal{F}$. Relations between terms can be established by the means of *equalities*. An *unconditional* equality is denoted by $s = t$, where s and t are two terms of same sort. Unconditional equalities and their negations are *literals*. A *clause* is a disjunction of literals. *Horn* clauses, consisting of clauses with at most one unconditional equality, are represented as implications. In its most usual form, $\neg e1 \vee \ldots \vee \neg e_n \vee e$ is a *conditional* equality, denoted by $e_1 \wedge \ldots \wedge e_n \Rightarrow e$, where e_i ($i \in [1..n]$) are *conditions* and e is the *conclusion*. Sometimes, we emphasize a

[1] The code of the Spike prover and the generated Coq scripts can be downloaded from http://code.google.com/p/spike-prover/.

particular condition e_i w.r.t. the other conditions Γ by writing $\Gamma \wedge e_i \Rightarrow e$. We denote by $e_1 \wedge \ldots \wedge e_n \Rightarrow$ an *impossible* set of conditions.

Equality reasoning may require transformations over equalities. A basic such transformation is the *substitution* operation which consists in simultaneous replacements of variables with terms. Formally, a substitution is represented as a finite mapping $\{x_1 \mapsto t_1; \ldots; x_n \mapsto t_n\}$, where $x_i \in \mathcal{V}$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. If σ is such a substitution, and t a term (resp. e an equality), then $t\sigma$ (resp. $e\sigma$) is an *instance* of t (resp. e). In the following, we assume that the variables from the replacing terms in σ are new w.r.t. the variables of t (resp. e). A term s *matches* a term t if there exists a (matching) substitution σ such that $s\sigma \equiv t$, where $\equiv$ is the *identity* relation. A *unifier* of two terms s and t is a substitution σ such that $s\sigma \equiv t\sigma$. In the rest of the paper, we will consider only the most general unifiers (mgu), and write $\sigma = mgu(s, t)$ whenever $s\sigma \equiv t\sigma$. Another kind of transformation operation is the replacement of a non-variable subterm of a term or equality by another term. The replaced subterm can be uniquely identified by its *position*. If p is a position and e (resp. t) an equality (resp. term), its subterm at position p is denoted by e_p (resp. t_p). Formally, $e_p[s]$ (resp. $t_p[s]$) states that s is a subterm of e (resp. t) at position p.

Any induction principle is based on an (induction) ordering. A *quasi-ordering* $\leq$ is a reflexive and transitive binary relation. The strict part of a quasi-ordering is called *ordering* and is denoted by $<$. We write $x >(\geq) y$ iff $y <(\leq) x$. An ordering $<$, defined over the elements of a nonempty set A, is *well-founded* if it is impossible to build an infinite strictly descending sequence $x_1 > x_2 > \ldots$ of elements of A. A binary relation R is *stable under substitutions* if whenever $s\,R\,t$ then $(s\sigma)\,R\,(t\sigma)$, for any substitution σ. Induction orderings can be defined over terms as well as equalities. An example of an induction ordering over terms is the recursive path ordering (for short, *rpo*), denoted by $\prec_{rpo}$, recursively defined from a well-founded ordering $<_{\mathcal{F}}$ over $\mathcal{F}$. If $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we write $t \prec_{rpo} s$ if $s \equiv f(s_1, \ldots, s_m)$ and i) either $s_i \equiv t$ or $t \prec_{rpo} s_i$ for some s_i, $i \in [1..m]$, or ii) $t \equiv g(t_1, \ldots, t_n), t_i \prec_{rpo} s$ for all i, $i \in [1..n]$, and either a) $g <_{\mathcal{F}} f$, or b) $f \equiv g$ and $(t_1, \ldots, t_n) \prec\!\!\prec_{rpo} (s_1, \ldots, s_n)$. $\prec\!\!\prec_{rpo}$ is the *multiset extension* of $\prec_{rpo}$. Given two multisets of terms A_1 and A_2, we write $A_1 \prec\!\!\prec_{rpo} A_2$ if, after the pairwise elimination of the identical terms from A_1 and A_2, $\forall s \in A_1$, $\exists t \in A_2$ such that $s \prec_{rpo} t$. An induction ordering over equalities, denoted by $\prec$, is defined as follows: $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s = t \prec s'_1 = t'_1 \wedge \ldots \wedge s'_m = t'_m \Rightarrow s' = t'$ if $\{s_1, t_1, \ldots, s_n, t_n, s, t\} \prec\!\!\prec_{rpo} \{s'_1, t'_1, \ldots, s'_m, t'_m, s', t'\}$. It can be shown that $\prec$ is well-founded and stable under substitutions.

We are interested in proving properties, or *conjectures*, about *conditional specifications* consisting in sets of conditional equalities defining (defined) function symbols, also referred to as *axioms*. An axiom $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s = t$ can be transformed into the (conditional) *rewrite rule* $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s \rightarrow t$ if $\{s_1, t_1, \ldots, s_n, t_n, t\} \prec\!\!\prec_{rpo} \{s\}$ and the variables of $s_1, t_1, \ldots s_n, t_n, t$ are among the variables of s. Given a substitution σ, a rewrite rule $a = b \Rightarrow l \rightarrow r$ and an equality e such that $e[l\sigma]_u$, a *rewrite operation* replaces $e[l\sigma]_u$ by $e[r\sigma]_u$, then enriches the conditions of $e[r\sigma]_u$ with the new conditions $a\sigma = b\sigma$. A *rewrite system* $\mathcal{R}$ consists of a set of rewrite rules. The rewrite relation $\rightarrow_{\mathcal{R}}$ denotes rewrite operations only with rewrite rules from $\mathcal{R}$ and it can be shown well-founded. The equivalence closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\overset{*}{\leftrightarrow}_{\mathcal{R}}$.

The equality $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s = t$ is an *inductive theorem* of a set of axioms orientable into a rewrite system $\mathcal{R}$ if, for any of its ground instances $s_1\sigma = t_1\sigma \wedge \ldots \wedge s_n\sigma = t_n\sigma \Rightarrow s\sigma = t\sigma$, whenever $s_i\sigma \overset{*}{\leftrightarrow}_{\mathcal{R}} t_i\sigma$, for each $i \in [1..n]$, we have $s\sigma \overset{*}{\leftrightarrow}_{\mathcal{R}} t\sigma$. *Tautologies* are inductive theorems either of the form $\wedge_i s_i = t_i \Rightarrow t = t$ or equalities for which the conclusion is among the conditions. An equality e_1 is *subsumed* by another equality e_2 if there exists a substitution σ such that $e_2\sigma$ is a *sub-clause* of e_1, i.e. e_1 is of the form $e_2\sigma \vee \Gamma$.

3 Noetherian Induction for Implicit Induction Proofs

Noetherian induction is a widely used induction principle. In its most general form, it allows to prove the validity of a property ϕ for all elements of a potentially infinite poset $(S, <)$, provided that $<$ is a well-founded ordering. The powerful of the principle consists in the possibility to use in the proof of any $\phi(m)$, with $m \in S$, the assumption that $\phi(k)$ is true, for all $k \in S$ smaller than m. Formally,

$$(\forall m \in S, (\forall k \in S, k < m \Rightarrow \phi(k)) \Rightarrow \phi(m)) \Rightarrow \forall m \in S, \phi(m)$$

The soundness of the principle is guaranteed by the well-foundedness property of $<$. The formulas $\phi(k)$, with $k < m$, can safely participate to the proof of $\phi(m)$ as *induction hypotheses*.

The Noetherian induction principle can be used to prove properties about conditional specifications if S is a set of equalities, $\phi(x) = x$ for all $x \in S$, and the well-founded ordering is $\prec$, as defined in the previous section. In this case, the Noetherian induction principle becomes a contrapositive version of the 'Descente Infinie' induction principle we have previously used in [20,21,22,23]:

$$(\forall \psi \in S, (\forall \rho \in S, \rho \prec \psi \Rightarrow \rho) \Rightarrow \psi) \Rightarrow \forall \gamma \in S, \gamma$$

This means that, in the proof of any equality, one can use smaller equalities.

To prove that a set of conditional equalities are inductive theorems w.r.t. a rewrite system $\mathcal{R}$, we will consider a simplified version of the Spike inference system [5,3], represented in Fig. 1, that implements this particular case of Noetherian induction in order to perform implicit induction proofs. The set S will consist of all ground instances of the conjectures encountered in the proof of the inductive theorems, defined as follows.

An implicit induction *proof* represents a finite sequence of *states* $(E^0, \emptyset) \vdash (E^1, H^1) \vdash \ldots \vdash (\emptyset, H^n)$, where E^i $(i \in [0..n-1])$ are multisets of conjectures and H^i $(i \in [1..n])$ are multisets of *premises* made of previously treated conjectures. A *step* between two states is performed by the application of one of the Spike *inference rules*. Each rule transforms one of the conjectures, called *current* conjecture, into a potentially empty set of new conjectures. GENERATE applies on equalities incorporating subterms that can unify with some lhs of the rewrite rules from $\mathcal{R}$. Such a subterm should be tested for unification against all the rewrite rules from $\mathcal{R}$, each mgu σ playing the role of *test* substitution. The resulted set of test substitutions is expected to be complete. The method for computing the test substitutions has been

GENERATE: $(E \cup \{e[t]_p\},\ H) \vdash (E \cup (\cup_\sigma \{e\sigma[r\sigma]_p \text{ enriched with cond. } a\sigma = b\sigma\})),\ H \cup \{e\})$
if $a = b \Rightarrow l \to r \in \mathcal{R}$, $\sigma = mgu(l, t)$.

TOTAL CASE REWRITING: $(E \cup \{e[t]_p\},\ H) \vdash (E \cup (\cup_\sigma \{\text{cond. } a\sigma = b \text{ added to } e[r\sigma]_p\})), H)$
if $a = b \Rightarrow l \to r \in \mathcal{R}$, $l\sigma \equiv t$, b is either $True$ or $False$.

(UNCONDITIONAL) REWRITING: $(E \cup \{e\},\ H) \vdash (E \cup \{e'\},\ H)$
if $e \to_{\mathcal{R} \cup \mathcal{L} \cup (H \cup E)_{\prec e}} e'$.

AUGMENTATION: $(E \cup \{\Gamma \wedge l \Rightarrow s = t\},\ H) \vdash (E \cup \{\Gamma \wedge \wedge_k l'_k \Rightarrow s = t\},\ H)$
if, for each k, it exists a lemma $u \Rightarrow v$ and a substitution σ such that
$u\sigma \equiv l$, $v\sigma \equiv l'_k$ and $l'_k \prec l$.

TAUTOLOGY: $(E \cup \{e\},\ H) \vdash (E,\ H)$
if e is a tautology.

SUBSUMPTION: $(E \cup \{e\},\ H) \vdash (E,\ H)$
if e is subsumed by some conjecture from $H \cup \mathcal{L}$.

Fig. 1. A simplified version of the Spike inference system.

presented in [3] and is different from the 'test set' method from [5]. Finally, the current conjecture is added to the set of premises. TOTAL CASE REWRITING can apply on equalities with subterms that are matched by some lhs of rewrite rules from $\mathcal{R}$. As for the unification test, the matching test should be done against all the rewrite rules from $\mathcal{R}$. The rewrite rules have the conditions restricted to the form $a = True$ or $a = False$ in order to simplify the certification process [23]. REWRITING rewrites an equality e with rewrite rules from $\mathcal{R}$, lemmas $\mathcal{L}$ consisting of previously proved conjectures, or smaller instances of premises H and conjectures E from the current state, denoted by $(H \cup E)_{\prec e}$. AUGMENTATION replaces one condition of the current conjecture with smaller conditions. The fact that the replaced condition implies the smaller conditions is stated by lemmas. TAUTOLOGY deletes tautologies. The last rule, SUBSUMPTION, deletes the equalities subsumed by either lemmas or premises. The *proof strategy* indicating the application order of the Spike rules is the following: on a given conjecture, TAUTOLOGY is firstly tried, then REWRITING followed by AUGMENTATION, SUBSUMPTION and TOTAL CASE REWRITING. When none of these rules works, GENERATE is finally applied.

The soundness property of the Spike inference system states that whenever a proof is derived then the initial conjectures are inductive consequences of the axioms. The inference system implements the 'Descente Infinie' principle such that whenever a current conjecture from a proof step can be instantiated to a false ground formula, there is another conjecture in the proof that can be instantiated to a smaller false ground formula, under the following conditions: i) the rewrite system $\mathcal{R}$ respects some syntactic criteria to ensure its *coherence*, i.e. an equality and its negation cannot be simultaneously consequences of the axioms, and ii) *completeness*, i.e. each function is defined in any point of the domain. In [19], we have proposed a methodology for

a modular checking of the soundness of implicit induction inference systems. The heart of the methodology is a very general inference system consisting only of two rules, AddPremise and Simplify, which has been shown sound, like any system built from instantiations of its rules. In our case, it can be shown that Generate is an instance of AddPremise, and the other rules are instances of Simplify.

4 Case Study: Validation of a Conformance Algorithm for the ABR Protocol

Available Bit Rate (ABR) [9] is one of the protocols for the ATM networks that fit well to manage the data rates between several applications sharing simultaneously a common physical link. What distinguishes ABR is its flexibility: the provider guarantees a minimum rate to the user, but the rate can fluctuate over time: it can increase if resources are available and should decrease if the network is congestioned. The rate management is a complex task for the provider. Firstly, the user is informed by the new rate values, then the user has to adapt the rates of its applications to the new values. Finally, the provider should test that the user rates are *conform* to a value which may vary in time, called *conformity value*, computed by the conformance algorithm running on a device positioned between the provider and the user. The new rate values arrive from the provider to the device by the means of Resource Management (RM-) cells. The rate and the arrival time of RM-cells are stored in a buffer such that the conformity value, at a given time, will depend on the rate values and the arrival time of the stored RM-cells.

There are several conformance algorithms, but here we will mention only two: Acr and Acr1. Both of them are ideal since they assume that the buffer can store an infinite number of RM-cells in order to give optimal conformity control from the user's point of view, i.e. the users are informed a) as soon as possible when the rate for their applications can augment, and b) as late as possible when the rate should decrease. Acr [4] has been standardized by the ATM Forum and considered as a landmark for the other algorithms. Defined later, Acr1 [14] is more efficient since it provides the conformity value quasi-instantaneously. The idea is to schedule into the future the rate of the RM-cell such that most of the computation is done at the time when an RM-cell arrives to the device and not when the conformity value is really needed. More exactly, at a given time t, the conformity value will be the rate of the first cell from the scheduled cell buffer, assuming that all the cells scheduled in the past (w.r.t. t) can be ignored or simply deleted.

Formally, a (RM- or scheduled) cell is a pair of naturals (t, er), where t is the (arrival or scheduled) time of the cell and er is its rate. A buffer is a list of cells $[(t_1, er_1), (t_2, er_2), \ldots, (t_n, er_n), \ldots]$. The RM-cell buffer is *time-decreasing*, i.e. $t_1 \geq t_2 \geq \ldots \geq t_n \geq \ldots$. In [15,17], Acr1 has been shown equivalent to Acr, i.e. for any configuration of the RM-cell buffer and any time t, both algorithms compute the same conformity value. Before proving in the next subsection one of the lemmas from the 'equivalence' proof, we introduce the following functions: i) $time(c)$ - returns the time value of the cell c, ii) $timel(l)$ - gives the time value of the first cell from the buffer l, iii) $le(n_1, n_2)$ - tests if the natural n_1 is less than or equal to the natural n_2, iv) $sortedT(l)$ - returns true if the buffer l is time-decreasing, and v) $insAt(l, t, e)$ -

deletes all the cells from the head of the buffer l having time values greater than t, then adds a new cell with the rate e and time t to the head of the buffer.

4.1 The Spike Specification

We will use the sort NAT to represent naturals, $BOOL$ for booleans, OBJ for cells, and $PLAN$ for lists of cells. The constructor symbols for the sorts NAT, $BOOL$, OBJ and $PLAN$, together with their profiles, are: $0 : NAT$, $S : NAT \to NAT$, $True : BOOL$, $False : BOOL$, $C : NAT\ NAT \to OBJ$, $Nil : PLAN$, and $Cons : OBJ\ PLAN \to PLAN$.

The defined functions and their defining axioms are:

$time : OBJ \to NAT$
[105] $time(C(u_1, u_2)) = u_1$

$timel : PLAN \to NAT$
[98] $timel(Nil) = 0$
[99] $timel(Cons(u_1, u_2)) = time(u_1)$

$le : NAT\ NAT \to BOOL$
[102] $le(0, u_1) = True$
[103] $le(S(u_1), 0) = False$
[104] $le(S(u_1), S(u_2)) = le(u_1, u_2)$

$sortedT : PLAN \to BOOL$
[107] $sortedT(Nil) = True$
[108] $sortedT(Cons(u_1, Nil)) = True$
[109] $le(u_1, u_2) = True \Rightarrow sortedT(Cons(C(u_2, u_3), Cons(C(u_1, u_4), u_5))) = sortedT(Cons(C(u_1, u_4), u_5))$
[110] $le(u_1, u_2) = False \Rightarrow sortedT(Cons(C(u_2, u_3), Cons(C(u_1, u_4), u_5))) = False$

$insAt : PLAN\ NAT\ NAT \to PLAN$
[130] $insAt(Nil, u_1, u_2) = Cons(C(u_1, u_2), Nil)$
[131] $le(time(u_1), u_2) = True \Rightarrow insAt(Cons(u_1, u_3), u_2, u_4) = Cons(C(u_2, u_4), Cons(u_1, u_3))$
[132] $le(time(u_1), u_2) = False \Rightarrow insAt(Cons(u_1, u_3), u_2, u_4) = insAt(u_3, u_2, u_4)$

The axioms can be oriented from left to right using the *rpo* ordering based on the following precedence over the function symbols: $0 <_F S <_F True <_F False <_F C <_F Nil <_F Cons <_F time <_F timel <_F le <_F sortedT <_F insAt$. For sake of simplicity, the axioms are referred to by unique identifiers that prefix them, like any conditional equality from the next section.

5 An Example of Implicit Induction Proof

The conjecture [301] $sortedT(u_1) = True \Rightarrow sortedT(insAt(u_1, u_2, u_3)) = True$, denoted by `sorted_insat1` in [17], will be proved with the inference system from Fig. 1 and the axioms from subsection 4.1, using the lemmas

[159] $le(u_1, u_2) = False \ \wedge \ le(u_1, u_2) = True \Rightarrow$
[196] $sortedT(Cons(u_1, u_2)) = True \Rightarrow sortedT(u_2) = True$
[226] $sortedT(Cons(u_1, u_2)) = True \Rightarrow le(timel(u_2), time(u_1)) = True$
[268] $sortedT(u_1) = True \ \wedge \ le(timel(u_1), u_2) = True \Rightarrow$
$sortedT(Cons(C(u_2, u_3), u_1)) = True$

The proof starts with the state $(\{[301]\}, \emptyset)$. The conjecture [301] cannot be simplified or deleted, so the only applicable rule is GENERATE. Firstly, the subterm $insAt(u_1, u_2, u_3)$ is unified with the lhs of the axioms [130], [131] and [132] using the test substitutions $\{u_1 \mapsto Nil\}$ (for [130]) and $\{u_1 \mapsto Cons(u_5, u_6)\}$ (for [131] and [132]). Since u_5 is an OBJ variable, it can be expanded to $C(u_8, u_9)$. Then, the resulted instances of [301] are rewritten with the corresponding axioms, respectively, to

[328] $sortedT(Nil) = True \Rightarrow sortedT(Cons(C(u_2, u_3), Nil)) = True$
[334] $sortedT(Cons(C(u_8, u_9), u_6)) = True \ \wedge \ le(time(C(u_8, u_9)), u_2) =$
$True \Rightarrow sortedT(Cons(C(u_2, u_3), Cons(C(u_8, u_9), u_6))) = True$
[340] $sortedT(Cons(C(u_8, u_9), u_6)) = True \ \wedge \ le(time(C(u_8, u_9)), u_2) =$
$False \Rightarrow sortedT(insAt(u_6, u_2, u_3)) = True$

The next proof state is $(\{[328], [334], [340]\}, \{[301]\})$. Each of the conjectures are rewritten with REWRITING as follows: firstly, the term $sortedT(Cons(C(u_2, u_3), Nil))$ from [328] to $True$ by the axiom [108], then $time(C(u_8, u_9))$ from [334] and [340] to u_8 by [105]. The new conjectures are:

[343] $sortedT(Nil) = True \Rightarrow True = True$
[346] $sortedT(Cons(C(u_8, u_9), u_6)) = True \ \wedge \ le(u_8, u_2) = True \Rightarrow$
$sortedT(Cons(C(u_2, u_3), Cons(C(u_8, u_9), u_6))) = True$
[349] $sortedT(Cons(C(u_8, u_9), u_6)) = True \ \wedge \ le(u_8, u_2) = False \Rightarrow$
$sortedT(insAt(u_6, u_2, u_3)) = True$

The conjecture [343] is deleted by TAUTOLOGY. Notice that each of the remaining conjectures are governed by the condition $sortedT(Cons(C(u_8, u_9), u_6)) = True$, so AUGMENT can be applied with the lemmas [196] and [226], to yield, respectively

[371] $le(u_8, u_2) = True \ \wedge \ sortedT(u_6) = True \ \wedge \ le(timel(u_6), time(C(u_8,$
$u_9))) = True \Rightarrow sortedT(Cons(C(u_2, u_3), Cons(C(u_8, u_9), u_6))) = True$
[396] $le(u_8, u_2) = False \ \wedge \ sortedT(u_6) = True \ \wedge \ le(timel(u_6),$
$time(C(u_8, u_9))) = True \Rightarrow sortedT(insAt(u_6, u_2, u_3)) = True$

The term $time(C(u_8, u_9))$ from the newly added conditions is again rewritten to u_9 by [105]. Now, the new set of conjectures consists of:

[374] $le(u_8, u_2) = True \ \wedge \ sortedT(u_6) = True \ \wedge \ le(timel(u_6), u_8) = True \Rightarrow$
$sortedT(Cons(C(u_2, u_3), Cons(C(u_8, u_9), u_6))) = True$

[399] $le(u_8, u_2) = False \ \wedge \ sortedT(u_6) = True \ \wedge \ le(timel(u_6), u_8) = True \Rightarrow sortedT(insAt(u_6, u_2, u_3)) = True$

The conjecture [399] is deleted by SUBSUMPTION with the premise [301] using the substitution $\{u_1 \mapsto u_6\}$. TOTAL CASE REWRITING is applied to the remaining conjecture [374] on the term $sortedT(Cons(C(u_2, u_3), Cons(C(u_8, u_9), u_6)))$ using the axioms [109] and [110], to give

[441] $le(u_8, u_2) = True \ \wedge \ sortedT(u_6) = True \ \wedge \ le(timel(u_6), u_8) = True \ \wedge \ le(u_8, u_2) = True \Rightarrow sortedT(Cons(C(u_8, u_9), u_6)) = True$

[445] $le(u_8, u_2) = True \ \wedge \ sortedT(u_6) = True \ \wedge \ le(timel(u_6), u_8) = True \ \wedge \ le(u_8, u_2) = False \Rightarrow False = True$

[441] is subsumed by the lemma [268] using the substitution $\{u_1 \mapsto u_6; u_2 \mapsto u_8; u_3 \mapsto u_9\}$. Similarly, [445] is subsumed by [159] with $\{u_1 \mapsto u_8\}$.

The proof of `sorted_insat1` ends successfully into the state $(\emptyset, \{[301]\})$.

6 Certifying Implicit Induction Proofs

The implicit induction proof from the previous section can be automatically checked for soundness using the certified reasoning environment provided by Coq [25]. The certification approach is based on the Noetherian induction principle from Section 3.

The Spike specification is firstly translated into a Coq script by the Spike prover before dealing with the proof part. The sorts and the function definitions are manually translated by the user, the following code being inlined in the Spike specification. Notice that only the types **OBJ** and **PLAN** have been defined by the user, **nat** and **bool** being predefined data structures in Coq.

```
Inductive OBJ:Set := C: nat → nat → OBJ.
Inductive PLAN:Set := Nil | Cons: OBJ → PLAN → PLAN.

Fixpoint le (m n:nat): bool :=
match m,n with
| 0,_⇒ true
| S_, 0 ⇒ false
| S x, S y ⇒ le x y
end.

Definition timel (o:PLAN): nat :=
match o with
| Nil ⇒ 0
| Cons o p ⇒ time o
end.
```

```
Definition time (o:OBJ) : nat
:=
match o with
| C t e ⇒ t
end.

Fixpoint insAt (p:PLAN) (t e:nat) :
PLAN :=
match p with
| Nil ⇒ Cons(C t e) Nil
| Cons o pg ⇒
  match le (time o) t with
  | true ⇒ Cons (C t e) (Cons o pg)
  | false ⇒ insAt pg t e
  end
end.
```

```
Fixpoint sortedT (p:PLAN) : bool
:=
match p with
| Nil ⇒ true
| Cons (C t1 e1) p1 ⇒
  match p1
  with
  | Nil ⇒ true
  | Cons (C t2 e2) p2 ⇒
     match le t2 t1 with
       | true ⇒ sortedT p1
       | false ⇒ false
     end
  end
end.
```

The Spike induction ordering is syntactic and exploits the tree representation of terms. In Coq, the syntactic representation is made explicit by abstracting the Coq terms into terms built from a term algebra provided by COCCINELLE [6]. Some information has to be fed in order to create operational term algebras, for example the set of function symbols. This information can be automatically inferred, as the rest of the transformations: each function symbol f is translated into `id_f`.

```
Inductive symb : Set :=
  | id_0 | id_S | id_true | id_false | id_C | id_Nil | id_Cons | id_le
  | id_time | id_time1 | id_sortedT | id_insAt.
```

The type of abstracted terms is defined as `Inductive` **term** : `Set :=` | `Var : variable` → **term** | `Term : symb` → `list` **term** → **term**. Any Coq term can be abstracted by *model* functions. They are defined by the user for each type.

```
Fixpoint model_nat (n:nat) : term
:=
match n with
| 0 ⇒ Term id_0 nil
| S n' ⇒ Term id_S ((model_nat n')::nil)
end.

Definition model_OBJ (o:OBJ) : term
:=
match o with
| C x y ⇒ Term id_C
   ((model_nat x)::(model_nat y)::nil)
end.
```

```
Definition model_bool (b:bool) :
term :=
match b with
| true ⇒ Term id_true nil
| false ⇒ Term id_false nil
end.

Fixpoint model_PLAN (p:PLAN) :
term :=
match p with
| Nil ⇒ Term id_Nil nil
| Cons o p ⇒ Term id_Cons ((
model_OBJ o)::(model_PLAN p)::nil)
end.
```

A Spike variable x of sort s can be transformed into the Coq variable x of type s', where s' is the Coq type corresponding to s. A non-variable term $f(t_1, \ldots, t_n)$ can be recursively transformed into the Coq term $(f\ t'_1 \ldots t'_n)$, where $t'_1, \ldots, t'_n$ are the transformations of $t_1, \ldots t_n$, respectively. The equality $e_1 \wedge \ldots e_n \Rightarrow e$ can be automatically translated into the Coq formula $\forall \overline{x}, e'_1 \rightarrow \ldots \rightarrow e'_n \rightarrow e'$, where $\overline{x}$ is the vector of all variables from the equality and $e'_1, \ldots, e'_n, e'$ are the equalities issued from the Coq transformations on the lhs and rhs of $e_1, \ldots, e_n, e$, respectively.

The proof part of the translation can be generated completely automatically. In order to perform induction reasoning, we explicitly pair any Coq formula F produced by the above transformations with a comparison weight $W(F)$ such that a formula F_1 is smaller than a formula F_2 if $W(F_1) \prec W(F_2)$, where $\prec$ is a well-founded and stable under substitutions ordering over weights. In our case, the weight of a formula is given by the list of the COCCINELLE terms abstracting the terms the formula is built from, as shown in [23]. The relation between a Coq formula and its weight should be stable under substitutions. This property is achieved if the common variables are factorized using functionals of the form $(\texttt{fun}\ \overline{x} \Rightarrow (F, W))$, where F is the Coq formula translating the Spike conjecture, W its weight, and $\overline{x}$ the vector of the common variables. The functional's type is associated to a proof and has to be general enough to represent all the conjectures from the proof. The type for the functionals from the proof of `sorted_insat1` and labelled as [301] in the previous section, an example of functional corresponding to [301], and all the functionals are:

```
Definition type_LF_301 := PLAN → nat → nat → nat → nat → (Prop × (list term)).
Definition F_301 : type_LF_301:= (fun u1 u2 u3 _ _ ⇒ ((sortedT u1) = true →
(sortedT (insAt u1 u2 u3)) = true, (Term id_sortedT ((model_PLAN u1)::nil))::(Term
id_true nil)::(Term id_sortedT ((Term id_insAt ((model_PLAN u1):: (model_nat u2)::
(model_nat u3)::nil))::nil))::(Term id_true nil)::nil)).
Definition LF_301 := [F_301, F_328, F_343, F_334, F_340, F_346, F_371, F_349,
F_396, F_374, F_441, F_445].
```

The Noetherian induction principle is represented as a Coq section, parameterized by four variables and two hypotheses. The variables must be specified and the hypotheses proved at each application of the theorem wf_subset. In our case, the variable T will correspond to $(\texttt{Prop} \times (\textbf{list term}))$, R to an ordering on pairs, and wf_R to the lemma stating the well-foundedness of the ordering.

```
Section wf_subset.

  Variable T : Type.
  Variable R : T → T → Prop.
  Hypothesis wf_R: well_founded R.
  Variable S : T → Prop.
  Variable P : T → Prop.
  Hypothesis S_acc : ∀ x, S x →
(∀ y, S y → R y x → P y) → P x.

  Theorem wf_subset: ∀ x, S x → P x.
  Proof.
  intro z; elim (wf_R z).
  intros x H1x H2x H3x.
  apply S_acc;
  intros y H1y H2y; apply H2x;
   trivial.
  Qed.
End wf_subset.
```

The main lemma, denoted by main_301, states that all formulas from the functionals of LF_301 are valid, assuming that for each formula one can use any smaller formula as induction hypothesis. The functions fst and snd return the first and the second projections of a pair, respectively, and less is the ordering over weights, presented in [23] and shown well-founded and stable under substitutions.

Lemma main_301 : $\forall$ *F*, In *F* LF_301 $\rightarrow$ $\forall$ *u1*, $\forall$ *u2*, $\forall$ *u3*, $\forall$ *u4*, $\forall$ *u5*, ($\forall$ *F'*, In *F'* LF_301 $\rightarrow$ $\forall$ *e1*, $\forall$ *e2*, $\forall$ *e3*, $\forall$ *e4*, $\forall$ *e5*, less (snd (*F' e1 e2 e3 e4 e5*)) (snd (*F u1 u2 u3 u4 u5*)) $\rightarrow$ fst (*F' e1 e2 e3 e4 e5*)) $\rightarrow$ fst (*F u1 u2 u3 u4 u5*).

The heart of its proof consists of a case analysis on the functionals from LF_301. As in [23], the associated formulas are proved using one-to-one translations of the corresponding Spike inference steps from the implicit induction proof presented in Section 5. We will detail only the translations for some applications of AUGMENTATION and SUBSUMPTION rules. Similar translations for the applications of the other rules can be found in [23].

We will firstly consider the application of AUGMENTATION on [346]. In the current state of the proof, *F* is instantiated with F_346, the last condition of the lemma is denoted by *Hind*, and the variables from the conclusion renamed to correspond to the conjecture [346] from the Spike proof. [371] is the result of the augmentation operation. In the first step, we instantiate *F'* from *Hind* with F_371 and denote it by *H*. The variables of F_371 are renamed in *HFabs0* to correspond to [371] from the Spike proof. This instance can be used in the proof as long as it is smaller than [346], according to *H*. The comparison between their weights is performed automatically by the user-defined strategy *solve_rpo_mul*, once the weights have been normalized by *rewrite_model*. The strategy *trivial_in* tests that F_371 is indeed at the sixth position (starting from 0) in LF_301. In the last step, the instances of the lemmas [226] and [196] are added as conditions and `auto` finally establishes the logical equivalence between [371] and [346].

`assert` (*H* := *Hind* F_371). `assert` (*HFabs0*: fst (F_371 *u6 u2 u3 u8 u9*)). `apply` *H*. *trivial_in* 6. *rewrite_model*. *abstract solve_rpo_mul*.

specialize true_226 `with` (*u2* := *u6*) (*u1* := (C *u8 u9*)).
specialize true_196 `with` (*u2* := *u6*) (*u1* := (C *u8 u9*)). `auto`.

The translation of the SUBSUMPTION application on [445] with lemma [159] is simpler since it does not require weight comparisons. The subsuming instance of [159] is contradicted and `auto` tests afterwards if it is a sub-clause of [445]. The subsumption test performed by Spike assumes that the equality operator is symmetric, which is not the case for Coq. For example, `auto` fails if we try to subsume $a = b$ with $b = a$. One solution is to apply `symmetry` before `auto`.

specialize true_159 `with` (*u1* := *u8*) (*u2* := *u2*). `intro` *L*. *contradict L*.
(`auto` || `symmetry`; `auto`).

Once the main proof is finished, we define the set S_301 of all instances of the functionals from LF_301, then prove the theorem all_true_301 stating that the associated formulas are true. A critical step of this proof is the use of wf_subset with the variable S instantiated by S_301. Finally, we prove [301] in true_301.

Definition S_301 := fun $f \Rightarrow \exists\ F$, In F LF_301 $\wedge \exists\ e1, \exists\ e2, \exists\ e3, \exists\ e4, \exists\ e5, f = F\ e1\ e2\ e3\ e4\ e5$.

Theorem all_true_301: $\forall\ F$, In F LF_301 $\rightarrow \forall\ u1$: **PLAN**, $\forall\ u2$: **nat**, $\forall\ u3$: **nat**, $\forall\ u4$: **nat**, $\forall\ u5$: **nat**, fst $(F\ u1\ u2\ u3\ u4\ u5)$.

Theorem true_301: $\forall\ (u1$: **PLAN**$)\ (u2$: **nat**$)\ (u3$: **nat**$)$, (sortedT $u1$) = true $\rightarrow$ (sortedT (insAt $u1\ u2\ u3$)) = true.

The same inference system and proof strategy involved in the proof of sorted_insat1 have been applied to prove other Spike conjectures involved in the ABR proof from [17]. In Table 1, we give some statistics about the proofs that have been automatically certified by Coq. We firstly list the name of the conjectures, as denoted in [17]. Then, for the proof of each conjecture, we show how many times each of the Spike inference rules has been applied, the number of needed lemmas, the size of the list of functionals for the main proof, and the global time of the certification process (lemmas + conjecture). For each conjecture, the Spike proof and its Coq translation lasted less than one second. All tests have been done on a MacBook Air featuring a 2.13 GHz Intel Core 2 Duo processor and 4 GB RAM.

6.1 Improvements

We have also tested the toy examples from [23] and found out that the certification time has been considerably reduced. For example, the certification process concerning the proofs about the validity of the sorting algorithm is now 20 times faster. We list the main improvements that allowed us to achieve this performance.

Weaker conditions for weight comparisons. An important part of the certification time concerning the examples from [23] has been spent in doing arithmetic reasoning, mainly using the expensive omega tactic. After fruitful discussions with the developpers of COCCINELLE, we changed less to avoid the size comparisons for the terms defining the compared weights. The size of the Coq scripts was also dramatically reduced.

Functionals labeling. We discovered that the membership test of a functional in a list of functionals is costly since it requires complex unification operations. We have avoided this problem by explicitly labeling the functionals, as for the definition of F_301. In this way, the membership test is performed on labels. Lots of comparisons have been avoided by pointing out the exact position of the label in the list.

Other improvements include a better readability of the Coq scripts by the definition of tactics using Ltac [8], the redefinition of the term algebra, and the separation between the specification (static) and proof (dynamic) parts.

Parallelisation of the certification process. The one-to-one translations from the main lemma can be proved independently. We have tested for parallelisation the proof of progat_insat, the most complex ABR proof from Table 1, as

Table 1. Some statistics about the ABR proofs.

# Name	Taut.	Rew.	Aug.	Sub.	Case	Rew.	Gen.	Lemmas	LF	time (s)
1. `firstat_timeat`	2	12	0	4	2	2	1	17	3.06	
2. `firstat_progat`	2	13	0	4	2	2	1	18	3.14	
3. `sorted_sorted`	2	1	0	0	0	1	0	6	1.58	
4. `sorted_insat1`	7	15	2	5	1	5	4	12	6.76	
5. `sorted_insin2`	7	22	2	5	1	5	4	20	7.54	
6. `sorted_e_two`	2	1	0	0	0	1	0	6	1.57	
7. `member_t_insin`	1	12	0	15	11	7	2	52	12.89	
8. `member_t_insat`	1	6	0	8	7	4	2	24	5.99	
9. `member_firstat`	2	12	0	8	6	3	2	29	6.39	
10. `timel_insat`	3	7	0	0	0	1	0	11	2.02	
11. `erl_insin`	3	8	0	0	0	1	0	12	2.17	
12. `erl_insat`	3	7	0	0	0	1	0	11	2.02	
13. `erl_prog`	9	29	0	0	0	3	2	18	8.90	
14. `time_progat_er`	2	11	0	2	1	2	1	15	2.50	
15. `timeat_tcrt`	4	6	0	1	2	1	0	16	3.40	
16. `timel_timeat_max`	5	23	2	3	2	3	3	33	7.63	
17. `null_listat`	1	7	0	3	1	2	1	11	2.42	
18. `null_listat1`	2	0	0	0	0	1	0	4	1.38	
19. `cons_insat`	0	1	0	1	0	1	0	4	1.46	
20. `cons_listat`	2	0	0	0	0	1	0	4	1.39	
21. `progat_timel_erl`	7	27	2	3	4	2	3	33	7.50	
22. `progat_insat`	11	85	1	23	26	4	4	134	57.09	
23. `progat_insat1`	8	30	1	7	7	4	3	42	15.31	
24. `timel_listupto`	3	4	0	0	0	1	0	8	1.81	
25. `sorted_listupto`	10	21	3	4	2	6	4	26	9.79	
26. `time_listat`	3	11	0	3	3	2	1	22	5.09	
27. `sorted_cons_listat`	9	24	1	5	9	4	4	42	15.66	
28. `null_wind2`	1	0	0	1	1	2	1	8	3.66	
29. `timel_insin1`	1	9	0	2	1	2	1	12	2.8	
30. `null_listupto1`	1	0	0	0	0	1	0	4	1.38	
31. `erl_cons`	3	8	0	0	0	1	0	12	2.03	
32. `no_time`	2	19	0	6	4	2	2	29	7.56	
33. `final`	4	4	0	8	5	2	3	13	3.90	

follows. Firstly, we have manually split the list of 134 functionals in n adjacent partitions such that the formulas from each partition i ($i \in [1..n]$) are proved in a process p_i. Another process p_c will prove the main lemma using the previous results. Ideally, under the assumption that there is only one process allotted per processor and the generation of each process is done automatically and instantaneously, the total certification time is $max\{t(p_i) \mid i \in [1..n]\} + t(p_c)$, where $t(p)$ is the execution time of the process p. We have experimented with different split strategies and found out cases where the certification process of the main lemma of `progat_insat` is 5 times faster.

7 Conclusions and Future Work

We have proposed a new methodology for certifying implicit induction proofs. The underlying induction principle is an instance of Noetherian induction based on an ordering over equalities which represents a contrapositive version of the 'Descente Infinie' induction principle from [23]. The soundness arguments are now fully constructive since there is no need to use additional hypotheses like the 'excluded middle' axiom required in the proof-by-contradiction argumentation inherent to the 'Descente Infinie' proofs.

We have also proposed improvements to automatize and speed-up the certification process of implicit induction proofs in order to deal with industrial-size applications. Concerning the ABR proofs, they have been performed by a restricted version of the Spike inference system and their certification was established within seconds for most of them. Compared to the Spike proofs from [17], the arithmetic reasoning was simulated with lemmas instead of using the complex decision procedures integrated into Spike [1]. Generating *complete* translations, i.e. valid Coq scripts from any valid Spike proof steps, is challenging even for simple inference rules. For example, the symmetry problem related to SUBSUMPTION was fixed only for the conclusion and not for the conditions of subsuming equalities. Hopefully, the proof of the main lemma is modular, so the failure of any one-to-one translation does not affect the rest of the proof.

In the near future, we plan to devise complete translations of the Spike inference rules and automatize the parallelisation process. A challenge will be to translate and certify the proofs involved in the JavaCard platform validation [3].

Acknowledgements We thank the ProVal team from INRIA Saclay - Île-de-France research center for helpful suggestions about the efficient use of COCCINELLE. Special thanks are addressed to the reviewers for their useful remarks.

References

1. A. Armando, M. Rusinowitch, and S. Stratulat. Incorporating decision procedures in implicit induction. *J. Symb. Comput.*, 34(4):241–258, 2002.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. G. Barthe and S. Stratulat. Validation of the JavaCard platform with implicit induction techniques. In R. Nieuwenhuis, editor, *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2003.
4. A. Berger, F. Bonomi, and K. Fendick. Proposed TM baseline text on an ABR conformance definition. Technical Report 95-0212R1, ATM Forum Traffic Management Group, 1995.
5. A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.
6. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. *Frontiers of Combining Systems*, pages 148–162, 2007.
7. J. Courant. Proof reconstruction. Research Report RR96-26, LIP, 1996. Preliminary version.

8. D. Delahaye. A tactic language for the system Coq. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)*, pages 85–95, Reunion Island (France), November 2000. Springer.
9. ITU-T. Traffic control and congestion control in B ISDN. Recommandation I.371.1, 1997.
10. C. Kaliszyk. Validation des preuves par récurrence implicite avec des outils basés sur le calcul des constructions inductives. Master's thesis, Université Paul Verlaine - Metz, 2005.
11. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.
12. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system - release 3.12. Documentation and user's manual*. INRIA.
13. F. Nahon, C. Kirchner, H. Kirchner, and P. Brauner. Inductive proof search modulo. *Annals of Mathematics and Artificial Intelligence*, 55(1–2):123–154, 2009.
14. C. Rabadan and F. Klay. Un nouvel algorithme de contrôle de conformité pour la capacité de transfert 'Available Bit Rate'. Technical Report NT/CNET/5476, CNET, 1997.
15. M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of a generic incremental ABR conformance algorithm. Technical Report 3794, INRIA, 1999.
16. M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. In E. A. Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 344–357. Springer, 2000.
17. M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. *J. Autom. Reasoning*, 30(2):53–177, 2003.
18. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS prover guide - version 2.4*. SRI International, November 2001.
19. S. Stratulat. A general framework to build contextual cover set induction provers. *J. Symb. Comput.*, 32(4):403–445, 2001.
20. S. Stratulat. Automatic 'Descente Infinie' induction reasoning. In B. Beckert, editor, *TABLEAUX*, volume 3702 of *Lecture Notes in Artificial Intelligence*, pages 262–276. Springer, 2005.
21. S. Stratulat. 'Descente Infinie' induction-based saturation procedures. In *SYNASC '07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 17–24, Washington, DC, USA, 2007. IEEE Computer Society.
22. S. Stratulat. Combining rewriting with Noetherian induction to reason on non-orientable equalities. In A. Voronkov, editor, *Rewriting Techniques and Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 351–365. Springer Berlin, 2008.
23. S. Stratulat. Integrating implicit induction proofs into certified proof environments. In *Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 320–335, 2010.
24. S. Stratulat and V. Demange. Validating implicit induction proofs using certified proof environments. Poster session of 2010 Grande Region Security and Reliability Day, Saarbrucken, March 2010.
25. The Coq Development Team. The Coq reference manual - version 8.2. http://coq.inria.fr/doc, 2009.