Automated Verification by Induction with Associative-Commutative Operators

Narjes Berregeb[§] Adel Bouhoula[§‡] Michaël Rusinowitch[§]

[§] INRIA Lorraine & CRIN
Campus Scientifique, 615, rue du Jardin Botanique - B.P. 101
54602 Villers-lès-Nancy Cedex, France
E-mail:{berregeb, bouhoula, rusi}@loria.fr

[‡] Computer Science Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, California 94025, USA
E-mail: bouhoula@csl.sri.com

Abstract. Theories with associative and commutative (AC) operators, such as arithmetic, process algebras, boolean algebras, sets, ... are ubiquitous in software and hardware verification. These AC operators are difficult to handle by automatic deduction since they generate complex proofs. In this paper, we present new techniques for combining induction and AC reasoning, in a rewrite-based theorem prover. The resulting system has proved to be quite successful for verification tasks. Thanks to its careful rewriting strategy, it needs less interaction on typical verification problems than well known tools like NQTHM, LP or PVS . We also believe that our approach can easily be integrated as an efficient tactic in other proof systems.

1 Introduction

Powerful tools based on model checking have been developed for the verification of finite-state systems [6]. Their extensions to some classes of infinite-state systems has only produced moderate success. Therefore deductive methods offer a promising complementary approach especially for verifying parameterized components or systems involving infinite data-types. Besides, when a program or a circuit is not correct, more high-level information about how to correct it can be derived with deductive methods.

Effective verification with deductive techniques requires efficient primitive inference procedures in order to free the user from tedious low-level proof construction details. Rewriting is now widely recognized as an important technique for efficiency and is part of many systems. In this framework, the induction prover SPIKE [1] [3, 2], has been developed. It relies on implicit induction whose principle is to simulate induction by term rewriting. Given a theory presented by conditional equations, the prover instanciates some particular variables of a conjecture to be proved, called *induction variables*, by terms from a *test set*

[1] Spike is available by ftp from ftp.loria.fr in /pub/loria/protheo/softwares/Spike

which is a finite description of the model, then simplifies them by axioms, other conjectures or induction hypotheses. Every iteration generates new subgoals that are processed in the same way as the initial conjectures.

However, many theories of interest include AC operators, which are hard to handle since they cause divergence or generate complex proofs. To overcome this problem, we propose to have the AC axioms built in the inference mechanism of SPIKE. The advantage of our approach over other implicit induction techniques [5, 9], is that it does not use AC unification (which is doubly exponential) during the proof process, but only AC matching.

In this paper, we propose methods for automatically selecting the induction variables of a conjecture to be proved, and for constructing test sets in the case of an AC conditional theory. We present our proof procedure as an inference system based on new simplification techniques. This inference system is correct, refutationally complete (when the procedure stops with failure we can ensure that the given conjecture is wrong) under some reasonable restrictions on the initial AC conditional theory. These results have been implemented in the system SPIKE-AC, and computer experiments have shown the gain we obtain when handling AC operators by these techniques. In particular, the procedure has allowed us to prove directly theorems (for example, the correctness of a ripple carry adder) that require more interaction with other systems.

Overview on an example

To illustrate our approach, let us describe the correctness proof of a simple digital circuit. We consider a ripple carry adder (see figure 1), whose inputs are two bit-vectors $A = (A_0, A_1, \ldots, A_{n-1})$ and $B = (B_0, B_1, \ldots, B_{n-1})$, and a carry C_0. This circuit performs addition of A and B and the result is a bit vector $S = (S_0, S_1, \ldots, S_{n-1})$, and a carry C_n. This problem is easily specified with conditional rules, and the specification obtained reflects clearly the circuit description. The circuit function computing the sum of two bit-vectors A and B given a carry C_0, is $add(A, B, C_0)$. We define a mapping function *bvtonat* which transforms a bit vector into an integer. The constructor *bitv(x,y)* builds a new bit-vector by concatening x as the least significant bit of the vector y, and the constant *Btm* is the empty vector. The correctness theorem states that when given two bit-vectors of the same size as inputs, the resulting output is, up to conversion, the arithmetic sum of the inputs. The conjecture to be proved is:

$$size(x_1) = size(x_2) \Rightarrow bvtonat(add(x_1, x_2, False)) = bvtonat(x_1) + bvtonat(x_2)$$

Using our techniques described in section 5, the test set computed for the specification is: $\{Btm; bitv(True, x_1); bitv(False, x_1); 0; s(x_1); True; False\}$, where: *Btm*, $bitv(True, x_1)$ and $bitv(False, x_1)$ are of type *vect*, 0 and $s(x_1)$ are of type *nat*, *True* and *False* are of type *bool*. The next step consists in applying an induction on the *induction variables* (see section 4). Here, the variables x_1 and x_2 are replaced by elements of the test set (whose variables are renamed), and the instances obtained are simplified. We thus obtain 9 subgoals to be proved.

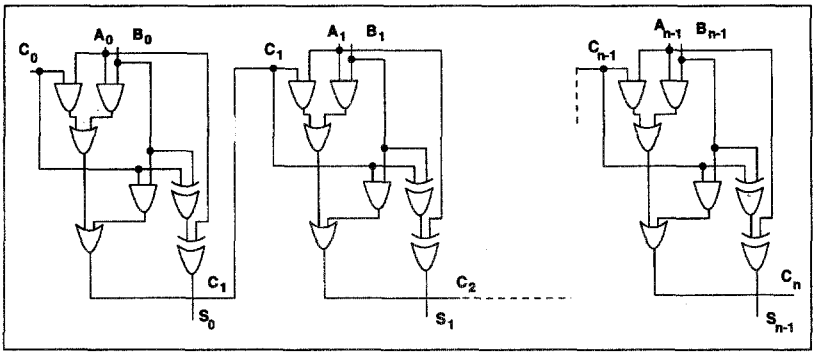

Fig. 1. Ripple carry adder

The simplification strategy may use axioms, other conjectures (even when they are not proved) and inductive hypotheses, provided they are smaller (w.r.t a well founded ordering on clauses). For example, the subgoal:

$$size(bitv(True, x_1)) = size(bitv(False, x_2)) \Rightarrow$$
$$s(bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False))) =$$
$$bvtonat(bitv(True, x_1)) + bvtonat(bitv(False, x_2))$$

is simplified using the axioms to:

$$size(x_1) = size(x_2) \Rightarrow$$
$$bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False)) =$$
$$bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_2) + bvtonat(x_2).$$

This simplification is not possible if we simply use the commutativity and associativity of + as lemmas, since then it would not be possible to derive a clause which is smaller than the starting one.
After simplifying and deleting the tautologies, only one subgoal remains to be proved:

$$size(x_1) = size(x_2) \Rightarrow$$
$$bvtonat(add(x_1, x_2, True)) + bvtonat(add(x_1, x_2, True)) =$$
$$s(s(bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_2) + bvtonat(x_2))).$$

This is the case for the addition of 2 bit-vectors when the carry is set to True. An induction on x_1 and x_2 must be applied. We obtain 9 subgoals to prove, and after simplification, 2 conjectures remain. The first one is:

$$size(x_1) = size(x_2) \Rightarrow$$
$$bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False)) +$$
$$bvtonat(add(x_1, x_2, False)) + bvtonat(add(x_1, x_2, False)) =$$
$$bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_1) + bvtonat(x_2) +$$
$$bvtonat(x_2) + bvtonat(x_2) + bvtonat(x_2)$$

It is reduced to a trivial identity using *inductive contextual rewriting* (see section 6) with the induction hypothesis:

$$size(x_1) = size(x_2) \Rightarrow bvtonat(add(x_1, x_2, False)) = bvtonat(x_1) + bvtonat(x_2)$$

The other remaining conjecture is simplified in the same way. Hence, all the subgoals are proved, and the initial goal is valid. The proof is completely automatic. Besides, it is easy to understand and very close to a mathematical proof. Note also that it does not use any specialized tactic nor any heuristic. The same conjecture has been proved with the NQTHM system [14], but then requires a non trivial generalisation of the input theorem. A proof was also done with PVS [7], but it uses a high-level user-defined proof strategy.

2 Basic concepts and notations

We assume that the reader is familiar with the basic concepts of term algebra, term rewriting, equational reasoning and mathematical logic. A many sorted signature σ is a pair $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S}$ is a set of *sorts* and $\mathcal{F} = F \cup F_{AC}$, where F and F_{AC} denote sets of function symbols. For short, a many sorted signature σ will simply be denoted by $\mathcal{F}$. The *variadic term algebra* is a generalisation of the term algebra, where AC functions symbols have a non fixed arity [12]. It allows us to express associative and commutative axioms by means of *flattening*. The variadic term algebra $TV(F, F_{AC}, \mathcal{X})$ over the signature $\mathcal{F}$ and the set of variables $\mathcal{X}$ is defined as the smallest set TV containing $\mathcal{X}$ such that:

- if $f \in F, arity(f) = n \geq 0$, and $t_1, \ldots, t_n \in TV$ then $f(t_1, \ldots, t_n) \in TV$.
- if $f \in F_{AC}, n \geq 2$, and $t_1, \ldots, t_n \in TV$ then $f(t_1, \ldots, t_n) \in TV$

In the following, $+$ will denote an AC symbol. *Flattening* a term consists of rewriting it to normal form w.r.t. the set of *flattening* rules: $f(x_1, \ldots, f(y_1, \ldots, y_r), z_1, \ldots, z_n) \rightarrow f(x_1, \ldots, y_1, \ldots, y_r, \ldots, z_1, \ldots, z_n)$ for all $f \in F_{AC}$. We denote by $flat(t)$, the term obtained by flattening t. A term s is *flattened* if $s = flat(s)$. We assume that we have a partition of $\mathcal{F}$ in two subsets, the first one, C, contains the *constructor symbols* and the second, D, is the set of *defined symbols*. We denote by $Var(t)$, the set of all variables appearing in t. A term is *linear* if all its variables occur only once in it. If $Var(t)$ is empty then t is a *ground* term. The set of all ground terms is $T(\mathcal{F})$. A *substitution* assigns terms of appropriate sorts to variables. Let t be a term, and η be a substitution, $t\eta$ is the flattened term obtained by applying η to t. The domain of η is defined by: $Dom(\eta) = \{x \mid x\eta \neq x\}$. If η applies every variable to a ground term, then η is a ground substitution. We denote by $\equiv$ the syntactic equivalence between objects. The smallest congruence generated by the equations $f(f(x, y), z) = f(x, f(y, z))$ and $f(x, y) = f(y, x)$ for all $f \in F_{AC}$ is denoted by $=_{AC}$. *Positions* in a term are defined in the same way than in [12]. The *replacement* of a term s by t at a position p is denoted by $s[p \leftarrow t]$. We assume that the term obtained is flattened. The term t/p is the subterm of t at position p. The notation $t[s]_p$ means that the term t contains a

subterm s at position p. We also denote by $t(p)$ the symbol of t at position p. For example, if $t = a+b+c$, then $t/\{2,3\} = b+c, t(\{2,3\}) = +, t/3 = c$. A position u in a term t such that $t(u) = x$ and $x \in \mathcal{X}$, is a *linear variable position* if x occurs only once in t, otherwise, u is a *non linear variable position.* A position u is a *strict position* of a term t if $t(u) \notin \mathcal{X}$, and $u = \varepsilon$ or $u = u'.i$ $(i \in \mathbb{N})$. The depth of a term t is defined as follows: $depth(t) = 0$ if t is a constant or a variable, otherwise, $depth(f(t_1, \ldots, t_n)) = 1 + max(depth(t_i))$. The strict depth of a term t, denoted by $sdepth(t)$, is the maximum of length of function positions in t.

A term s *matches* a term t if there exists a substitution σ such that $t =_{AC} s\sigma$; the term t is called an AC *instance* of s. A term t is AC *unifiable* with a term s, if there exists a substitution σ such that $t\sigma =_{AC} s\sigma$.

An ordering $\succ$ is AC *compatible* if $s' =_{AC} s$, $s \succ t$ and $t =_{AC} t'$ implies $s' \succ t'$. In the following, we suppose that $\succ$ is a transitive irreflexive relation on the set of terms, that is noetherian, monotonic ($s \succ t$ implies $w[s]_u \succ w[t]_u$), stable ($s \succ t$ implies $s\sigma \succ t\sigma$), AC compatible and satisfy the subterm property ($f(\cdots, t, \cdots) \succ t$). The multiset extension of $\succ$ will be denoted by $\gg$.

A *conditional* equation a formula of the following form: $a_1 = b_1 \wedge \cdots \wedge a_n = b_n \Rightarrow l = r$. It will be written $a_1 = b_1 \wedge \cdots \wedge a_n = b_n \Rightarrow l \to r$ and called a *conditional rule* if $\{l\sigma\} \gg \{r\sigma, a_1\sigma, b_1\sigma, \cdots, a_n\sigma, b_n\sigma\}$ for each substitution σ and every variable of the conditional equation occurs in l. The term l is the *left-hand side* of the rule. A rewrite rule $c \Rightarrow l \to r$ is *left-linear* if l is linear. A set of conditional rules is called a *rewrite system.* A constructor is *free* if it is not the root of a left-hand side of a rule. We denote by $lhss(R)$, the set of subterms of all left-hand sides of R. The number of elements of a set T is $card(T)$. A rewrite system R is *left-linear* if every rule in R is left-linear. We say that R is flattened if all its left-hand sides are flattened. Let f be an AC symbol, we denote by b_f the maximal arity of f in the left-hand sides of R. The depth (resp. strict depth) of a rewrite system R, denoted by $depth(R)$ (resp. $sdepth(R)$), is the maximum of the depths (resp. strict depth) of its flattened left-hand sides. We define $D(R)$ as $depth(R) - 1$ if $sdepth(R) < depth(R)$ and R is left-linear, otherwise $depth(R)$.

Let t be a flattened term, we write $t \to_R t'$ if there exists a conditional rule $\wedge_{i=1}^{n} a_i = b_i \Rightarrow l \to r$ in R, a position p and a substitution σ such that:

- $t/p =_{AC} l\sigma$, $t' =_{AC} t[p \leftarrow r\sigma]$.
- *for all* $i \in [1 \cdots n]$ there exists c_i, c_i' such that $a_i\sigma \to_R^* c_i$, $b_i\sigma \to_R^* c_i'$ and $c_i =_{AC} c_i'$.

where the reflexive-transitive closure of $\to$ is denoted by $\to^*$. In this case we say that the term t is *reducible*, otherwise, it is *irreducible.* From now on, we assume that there exists at least one irreducible ground term of each sort. We say that two terms s and t are *joinable*, if $s \to_R^* v$, $t \to_R^* v'$ and $v =_{AC} v'$. A term t is *inductively reducible* iff all its ground instances are reducible. A symbol $f \in \mathcal{F}$ is *completely defined* if all ground terms with root f are reducible. We say that R is *sufficiently complete* if all symbols in D are completely defined.

A *clause* C is an expression of the form: $\neg(s_1 = t_1) \vee \neg(s_2 = t_2) \vee \cdots \vee \neg(s_n = t_n) \vee (s_1' = t_1') \vee \cdots \vee (s_m' = t_m')$. We naturally extend the notion of *flat-*

tening, substitution, positions to clauses. Let H be a set of conditional equations and F_{AC} a set of AC symbols. The clause C is a *logical consequence* of H if C is valid in any model of $H \cup \{f(f(x,y),z) = f(x,f(y,z)),\ f(x,y) = f(y,x)$ for all $f \in F_{AC}\}$. This will be denoted by $H \models C$. We say that C is *inductively valid* in H and denote it by $H \models_{ind} C$, if for any ground substitution σ, *(for all* i $H \models s_i\sigma = t_i\sigma$ *) implies (there exists* j *such that* $H \models s'_j\sigma = t'_j\sigma$*)*. The rewrite system R is *ground convergent* if the terms u and v are joinable whenever $u, v \in T(F)$ and $R \models u = v$. In this paper, we suppose that all clauses and terms are flattened, and we denote by R a flattened rewrite system.

3 Induction schemes

To prove a conjecture by induction, the prover computes automatically an induction scheme, which consists of a set of variables on which induction is applied and a set of terms covering the possibly infinite set of irreducible ground terms.

Definition 3.1 *Given a term* t*, a set* $V \subseteq Var(t)$ *and a set of terms* T*, a* (V,T)*-substitution is a substitution of domain* V*, such that for all* $x \in V$*,* $x\sigma$ *is an element of* T *whose variables have been given new names.*

Definition 3.2 *An induction scheme* $\mathcal{I}$ *for a term* t *is a couple* (V,T)*, with* $V \subseteq Var(t)$ *and* $T \subseteq T(F, F_{AC}, \mathcal{X})$*, such that: for every ground irreducible term* s*, there exists a term* t *in* T *and a ground substitution* σ *such that* $t\sigma =_{AC} s$*.*

These induction schemes allow us to prove theorems by induction, by reasoning on the domain of irreducible terms rather than on the whole set of terms. However, they cannot be used to refute false conjectures. In the following, we refine induction schemes so that to be able, not only to prove conjectures, but also to refute the false ones.

Definition 3.3 *A term* t *is strongly irreducible if none of its subterms is an instance of a left-hand side of a rule in* R*.*

Definition 3.4 *A strong induction scheme* $\mathcal{I}$ *for a term* t *is an induction scheme* (V,T)*, where* V *is called the set of induction variables, and* T *is called the test set, such that: for each term* t *and* $\mathcal{I}$*-substitution* σ*, if* $t\sigma$ *is strongly irreducible, then there exists a ground substitution* τ *such that* $t\tau$ *is irreducible.*
An $\mathcal{I}$*-substitution is called* test substitution*.*

The next definition provides us with a criteria to reject false conjectures. Then, we show that strong induction schemes are fundamental for this purpose (see theorem 3.1)

Definition 3.5 *A clause* $\neg(s_1 = t_1) \vee \cdots \vee \neg(s_m = t_m) \vee (g_1 = d_1) \vee \cdots \vee (g_n = d_n)$ *is provably inconsistent with respect to* R *if there exists a test substitution* σ *such that:*

1. $\forall i \in [1 \cdots m] : s_i\sigma = t_i\sigma$ *is an inductive theorem w.r.t.* R*.*

2. $\forall j \in [1 \cdots n] : g_j\sigma \neq_{AC} d_j\sigma$, *and the maximal elements of* $\{g_j\sigma, d_j\sigma\}$ *w.r.t.*$\succ$ *are strongly irreducible.*

The next result shows that a provably inconsistent clause cannot be inductively valid w.r.t. R. This is proved by building a well-chosen ground instance of the clause which gives us a counterexample.

Theorem 3.1 *Suppose R is a ground convergent rewriting system. If a clause C is provably inconsistent, then C is not inductively valid w.r.t R.*

In the following sections, we propose methods for automatically computing each component of a strong induction scheme, that are, the induction variables and the test set.

4 Selecting induction variables

To prove a conjecture by induction, the prover selects automatically the induction variables of the conjecture where induction must be applied, then, instanciates them with terms of the test set. It is clear that the less induction variables we have, the more efficient the induction procedure will be.

To determine induction variables, the prover computes first the induction positions of the functions. These positions enable to decide whether a variable position of a term t is an induction variable or not. The induction positions computation is done only once and before the proof process. It is independent from the conjectures to be proved since it is based only on the given conditional theory.

Definition 4.1 *Let t be a term such that $t(\varepsilon) = f$ and $f \in \mathcal{F}$. A position $i \in \mathbb{N}$ is an inductive position of f in t if i is either a strict position in t, or a non linear variable position. We define $pos\_ind(f,t)$ as the set of inductive positions of f in t and $pos\_ind(f) = \bigcup_{t \in lhss(R)} pos\_ind(f,t)$.*

The idea is that a variable in a term t will be selected as an induction variable if it occurs below an inductive position. Hence, instantiating these variables may trigger a rewriting step. A problem happens with an AC symbol f, since the inductive positions of f can be permuted. For example, let $R = \{x + 0 + 0 \rightarrow 0\ ,\ x + 1 + 1 \rightarrow 0\}$, with $F_{AC} = \{+\}$ and $F = \{0, 1\}$. We have $pos\_ind(+) = \{2, 3\}$. Considering only y and z as induction variables, the proof of the conjecture $x + y + z = 0$ fails. However, it is an inductive theorem since all its ground instances are logical consequences of R.

This leads us to take all variables occuring under an AC symbol, as induction variables, so that to ensure the refutational completeness of our procedure, that is, whenever the proof of a clause finitely fails, we can ensure that it is not an inductive consequence of R. However, in order to make the proof process efficient, we have identified some cases where the number of induction variables to consider can be reduced while preserving refutational completeness.

For this purpose, for each $f \in F_{AC}$, we define the number $nb\_pos\_ind(f) = max_{t \in lhss(R)} card(pos\_ind(f,t))$. We denote by $var\_ind(t)$ the set of induction variables of t. The procedure computing induction variables is given in figure 2. We assume that the three predicates P_1, P_2, P_3 are defined as follows:

$P_1(f,R) \Leftrightarrow f$ is completely defined and $nb\_pos\_ind(f) = 1$
$P_2(f,R) \Leftrightarrow f$ is completely defined and $nb\_pos\_ind(f) > 1$
$P_3(f,R) \Leftrightarrow R$ is left-linear and for each $f(t_1,\ldots,t_n) \in lhss(R)$ there does not exist two non variable terms t_i, t_j which are AC unifiable

input: t **output:** $var\_ind(t)$ **init:** $Vind := \emptyset$
if t is a variable
then $Vind := \{t\}$
else for each position u in t such that $t(u) = f$ and $f \in F$ **do:**
$Vind := Vind \cup \{x \mid x$ appears at position $u.i$, and $i \in pos\_ind(f)\}$
endfor
for each $f \in F_{AC}$ in t **do:**
case 1: $P_1(f,R)$ and there is a variable x which is an argument of each occurrence of f in t: $Vind := Vind \cup \{x\}$
endcase 1
case 2: $P_2(f,R)$:
for each position u in t such that $t(u) = f$ **do:**
let $X_u = \{x \in \mathcal{X} \mid x$ appears at a position $u.i (i \in \mathbb{N})\}$
if $(X_u \cap Vind = \{x\})$ and $(\exists y \in X_u)$
then $Vind := Vind \cup \{y\}$
else if $(X_u \cap Vind = \emptyset)$ and $(X_u = \{x\})$
then $Vind := Vind \cup \{x\}$
else if $(X_u \cap Vind = \emptyset)$ and $(\{x,y\} \subseteq X_u)$
then $Vind := Vind \cup \{x,y\}$
endif
endif
endif
endcase 2
case 3: $P_3(f,R)$ and there is a variable x which is an argument of each occurrence of f in t: $Vind := Vind \cup \{x\}$
endcase 3
case 4: otherwise:
for each position u such that $t(u) = f$ **do:**
let $X_u = \{x \in \mathcal{X} \mid x$ appears at a position $u.i (i \in \mathbb{N})\}$
$Vind := Vind \cup X_u$
endfor
endcase 4
endfor
endif
return$(Vind)$

Fig. 2. Induction variables computation

Example 4.1 *Consider the following rewriting system R, with $F_{AC} = \{+, *\}$.*

$$R = \begin{cases} x + 0 \to x & x + s(y) \to s(x + y) \\ x * 0 \to 0 & x * s(y) \to x + x * y \\ exp(z, 0) \to s(0) & exp(z, s(n)) \to z * exp(z, n) \end{cases}$$

We have: $nb\_pos\_ind(+) = 1$, $nb\_pos\_ind(*) = 1$, $pos\_ind(exp) = 2$. A test set for R is $\{0, s(x)\}$. The conjecture to prove is $(x+y+z)*w = x*w+y*w+z*w$. If we take x, y, z, w as induction variables, we would obtain 16 lemmas to prove. Now, since $+$ and $*$ are completely defined AC operators, $nb\_pos\_ind(+) = 1$ and $nb\_pos\_ind(*) = 1$, we can choose $\{x, w\}$ or $\{y, w\}$ or $\{z, w\}$ as induction variables. Thus, we obtain only 4 lemmas to prove.

5 Computing test sets

The computation of test sets according to definition 3.4 relies on the inductive reducibility property [9], which is unfortunately undecidable in AC theories [10]. However we can use a semi-decision procedure that has proved to be quite useful for practical applications [11]. For the more restricted case where the rewrite system is left-linear and sufficiently complete, and the relations between constructors are equational we propose algorithms basically extending the equational case [9, 5] (see theorem 5.1). For the case where the rewrite system is not left linear but it is sufficiently complete over free constructors there is an easy algorithm to produce test-sets (see theorem 5.2).

We denote by $extension(t)$ the term obtained by replacing each subterm of t of the form $f(t_1, \ldots, t_{b_f+1})$ by $f(t_1, \ldots, t_{b_f+1}, x)$, where $f \in F_{AC}$ and x is a new variable. Given a set of terms T, $extension(T) = \bigcup_{t \in T} extension(t)$.

Theorem 5.1 *Let R be a left-linear conditional rewriting system. Let $T = \{t \mid t$ is a term of depth $\leq D(R)$ such that all variables occur at depth $D(R)$, and each AC operator has a number of arguments $\leq b_f + 1\}$. Let $T' = \{t \in T \mid t$ is not inductively reducible $\}$. Then $extension(T')$ is a test set for R.*

Example 5.1 *Let $F = \{0, 1, s\}$, $F_{AC} = \{+\}$ and*

$$R = \begin{cases} 0 + x \to x, \\ 1 + s(x) \to s(s(x)), \\ s(1) \to s(s(0)) \end{cases}$$

We have: $D(R) = 1$. By applying theorem 5.1, we obtain: $T' = \{0,\ 1,\ s(x),\ x + y,\ x + y + z\}$, $extension(T') = \{0,\ 1,\ s(x),\ x + y,\ x + y + z,\ x + y + z + t\}$, which can be simplified by deleting the subsumed terms and give the test set: $\{0,\ 1,\ s(x),\ x + y\}$.

A sort $s \in \mathcal{S}$ is said *infinitary* if there exists an infinite set of ground irreducible terms of sort $\mathcal{S}$. The next theorem provides a method for constructing test sets for non left-linear rewriting system with free constructors.

Theorem 5.2 *Let R be a conditional rewriting system. Suppose that R is sufficiently complete over free constructors. Then, the set T of all constructors terms of depth $\leq D(R)$ such that all variables are infinitary and occur at a depth $D(R)$, is a test set for R.*

6 Inference system

Our inference system rules (see figure 3) is based on a set of transition rules applied to (E, H), where E is the set of conjectures to prove and H is the set of induction hypotheses. The initial set of conditional rules R is oriented with a well founded and AC compatible ordering.

The inference system is constituted of two rules: *generation* and *simplification*. An *I-derivation* is a sequence of states: $(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots (E_n, H_n) \vdash_I \ldots$. We say that an I-derivation is *fair* if the set of persistent clauses $(\cup_i \cap_{j \geq i} E_j)$ is empty. An I-derivation fails when it is not possible to extend it by one more step and there remains conjectures to prove. We denote by $\prec_c$ a noetherian ordering on clauses, stable modulo AC, that extends $\prec$. In the following, W denotes a set of conditional equations which can be induction hypotheses or conjectures not yet proved. Let us now present briefly the rewriting techniques used by the prover. Inductive contextual rewriting is a generalization of both inductive rewriting [3] and contextual rewriting [15].

Definition 6.1 (Inductive contextual rewriting) *Given a clause C, we write:*

$$C \equiv \Delta \Rightarrow A \longmapsto_{R<W>} C' \equiv \Delta \Rightarrow A[u \leftarrow t\sigma]$$

if there exists $\delta \equiv \Gamma \Rightarrow s = t \in R \cup W$ and a position u in A such that:

- $A/u =_{AC} s\sigma$
- $C' \prec_c C$
- *if* $\delta \in W$ *then* $\delta\sigma \prec_c C$
- $R \cup W^{\prec_c} \models_{ind} \Delta \Rightarrow \Gamma\sigma$

where $W^{\prec_c} = \{\Phi \mid \Phi \in W \text{ and } \Phi \prec_c C\}$.

Example 6.1 *Let* $F_{AC} = \{+\}$ *and* $C \equiv (odd(1+1) = True \vee equal(1,1) = False \vee even(1+1) = True)$. *Suppose that we have an induction hypothesis:* $H \equiv (equal(x,y) = False \vee even(x+y) = True)$. *The inductive contextual rewriting of* C *by* H *gives:* $C' \equiv (odd(1+1) = True \vee equal(1,1) = False \vee True = True)$.

Inductive case rewriting provides us with a possibility to perform a case-based reasoning; it simplifies a conjecture with an axiom, a conjecture or an inductive hypothesis, provided it is smaller than the initial conjecture and the disjunction of all conditions is inductively valid.

Definition 6.2 (Inductive case rewriting) *Let G be the set* $\{< C[u \leftarrow d\sigma], P\sigma > |$ *there exists* $\mathcal{R} \equiv P \Rightarrow g \rightarrow d$ *in* $R \cup W$, *and a position* u *in* C *such that* $C/u =_{AC} g\sigma$, *and if* $\mathcal{R} \in W$, *then* $\mathcal{R} \prec_c C \}$. *If* $R \models_{ind} (\bigvee_{<C',P> \in G} P)$, *then* Inductive_case_rewriting(C,W) $= \{P \Rightarrow C' \mid < C', P > \in G\}$.

Generation: $(E \cup \{C\}, H) \vdash_I (E \cup E', H \cup \{C\})$
if $E' = \bigcup_\sigma simplify(C\sigma, H \cup E \cup \{C\})$, σ ranging over test substitutions of C

Simplification: $(E \cup \{C\}, H) \vdash_I (E \cup E', H)$
if $E' = simplify(C, H \cup E)$

Fig. 3. Inference system I

Definition 6.3 (simplify) *The procedure* simplify *is defined in the following way:*

simplify(C, W) =
  if *C is a tautology or subsumed by a clause of* $R \cup W$
  then $\emptyset$
  else if $C \longmapsto_{R<W>} C'$
    then $\{C'\}$
    else Inductive_case_rewriting(C,W)

The correctness of the inference system I is expressed by the following theorem:

Theorem 6.1 (Correctness) *Let* $(E_0, \emptyset) \vdash_I (E_1, H_1) \vdash_I \ldots$ *be a fair I-derivation. If it does not fail then* $R \models_{ind} E_0$.

Now, consider boolean specifications. To be more specific, we assume there exists a sort *bool* with two free constructors $\{true, false\}$. Every rule in R is of type: $\wedge_{i=1}^{n} p_i = p_i' \Rightarrow s \rightarrow t$ where for all i in $[1 \cdots n]$, $p_i' \in \{true, false\}$. Conjectures will be *boolean clauses*, i.e. clauses whose negative literals are of type $\neg(p = p')$ where $p' \in \{true, false\}$. If for all rules of form $p_i \Rightarrow f(t_1, \ldots, t_n) \rightarrow r_i$ whose left hand sides are identical up to a renaming μ_i, we have $R \models_{ind} \vee_i p_i\mu_i$, then f is weakly complete w.r.t R. We say that R is weakly complete if any function in $\mathcal{F}$ is weakly complete [1]. We can show that refutational completeness is also preserved in the AC case.

Theorem 6.2 (Refutational completeness) *Let R be a weakly complete and ground convergent rewrite system. Let* E_0 *be a set of boolean clauses. Then* $R \not\models_{ind} E_0$ *iff all fair derivations issued from* $(E_0, \emptyset)$ *fail.*

7 Conclusion

We have presented a new induction procedure for the associative and commutative theories. An advantage of this approach is that inference steps are performed in a homogeneous well-defined framework. Another important point is that our

procedure does not need AC unification like completion methods, but only AC matching. Our inference system is based on two rules: the generation rule which performs induction, and the simplification rule which simplifies conjectures by elaborated rewriting techniques. This system is correct and refutationally complete for boolean ground convergent rewrite systems under reasonable restrictions. In experiments, refutational completeness is particularly useful for debugging specifications. These results have been integrated in the prover SPIKE-AC, and interesting examples such as circuits verification have demonstrated the advantages of the approach.

References

1. A. Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170, December 1996.
2. A. Bouhoula, E. Kounalis, M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631-668, 1995.
3. A. Bouhoula, M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.
4. R. S. Boyer, J. S. Moore. *A Computational Logic Handbook*. 1988.
5. R. Bündgen, W. Küchlin. Computing ground reducibility and inductively complete positions. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, *LNCS* 355, pages 59–75, 1989.
6. J.R. Burch, E. M. Clarke, K.L. McMillan, D.L. Dill. Symbolic Model Checking: 10^{20} states and beyond. *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428-439, 1990.
7. D. Cyrluk, S. Rajan, N. Shankar, M. K. Srivas. Effective Theorem Proving for Hardware Verification. In K. Ramayya and K. Thomas, editors, *Theorem Provers in Circuit Design LNCS* 901, pages 203–222, 1994.
8. S. J. Garland, John V. Guttag. An overview of LP, the Larch Prover. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, *LNCS* 355, pages 137–151, 1989.
9. J.-P. Jouannaud, E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989.
10. D. Kapur, P. Narendran, D. J. Rosenkrantz, H. Zhang. Sufficient completeness, ground-reducibility and their complexity. *Acta Informatica*, 28:311–350, 1991.
11. E. Kounalis M. Rusinowitch. Reasoning with conditional axioms. *Annals of Mathematics and Artificial Intelligence*, (15):125–149, 1995.
12. C. Marché. *Réécriture modulo une théorie présentée par un système convergent et décidabilité du problème du mot dans certaines classes de théories équationnelles.* Th. univ., Université de Paris-Sud (France), 1993.
13. S. Owre, J.M. Rushby, N. Shankar. A prototype verification system. In D. Kapur, editor, *International Conference on Automated Deduction*, *LNAI* 607, pages 748–752, 1992.
14. L. Pierre. An automatic generalisation method for the inductive proof of replicated and parallel architetures. In K. Ramayya and K. Thomas, editors, *Theorem Provers in Circuit Design*, *LNCS* 901, pages 72–91, 1994.
15. H Zhang. Contextual rewriting in automated reasoning. *Fundamenta Informaticae*, (24):107–123, 1995.