Proving the Correctness of Algebraic Implementations by the ISAR System

Bernhard Bauer *, Rolf Hennicker**

* Institut für Informatik, Technische Universität München,
Arcisstr. 16, D-8000 München 2,
E-mail: bauer@informatik.tu-muenchen.de
** Institut für Informatik, Ludwig-Maximilians-Universität München,
Leopoldstr. 11b, D-8000 München 40,
E-mail: hennicke@informatik.uni-muenchen.de

Abstract. We present an interactive system, called ISAR, which provides an environment for correctness proofs of algebraic implementation steps. The correctness notion of implementation is based on behavioural semantics and the underlying proof procedure of the system is based on the principle of context induction (which is a particular instance of structural induction). The input of the ISAR system is an implementation step consisting of an abstract specification to be implemented, a concrete specification used as a basis for the implementation and an implementation construction. If all steps of the (interactive) proof procedure are performed the system has proved the correctness of the implementation step.

1 Introduction

Much work has been done in the field of algebraic specifications to provide formal concepts for the development of correct programs from given specifications. However, in order to be useful in practice, a formal theory of correct program development is not sufficient: Formal implementation notions should be supplied by appropriate proof methods and, even more important, by tools providing mechanical support for proving the correctness of implementation steps.

In this paper an interactive system for algebraic implementation proofs, called ISAR, is presented which sets out from the observational view of software development: The basic assumption is that a software product is a correct implementation if it satisfies the desired input/output behaviour, independently from the internal properties of a program which may not satisfy a given specification. This covers well known practical examples like the implementation of sets by lists (since lists do not satisfy the characteristic set equations but lists have the same behaviour as sets if only membership tests $x \in S$ are observable) or the familiar implementation of stacks by arrays with pointer (since arrays with pointer do not satisfy the usual stack equation $pop(push(x, s)) = s$ but they have the same behaviour as stacks if only the top elements of stacks are observed).

A formalization of this intuitive idea is presented in [Hennicker 90, 92] where an implementation relation for specifications is defined based on behavioural semantics

in the sense of [Nivela, Orejas 88], [Reichel 85]. In particular, in [Hennicker 90, 92] a proof theoretic characterization and a proof method, called *context induction*, is presented for proving behavioural implementation relations. The characterization of implementations says that a specification *SP1* is a *behavioural implementation* of a specification *SP* if and only if for all *observable contexts c* (over the signature of *SP*) and for all axioms t = r of *SP* the "observable" equations c[t] = c[r] are deducible from the axioms of the implementation *SP1* (for all ground instantiations over the signature of *SP*).
It is the basic idea of the ISAR system to prove this condition by context induction, i.e. by structural induction on the set of observable contexts, in order to show that SP1 is an implementation of SP. The underlying algorithm of the ISAR system providing a procedure for context induction proofs was developed in [Hennicker 92].
Usually implementations of an abstract specification are constructed on top of existing (concrete) specifications of standard data structures like lists, arrays, trees etc. In order to document the construction of the implementation, the input of the ISAR system is an *implementation step* which consists of three parts: an abstract specification SP-A to be implemented, a concrete specification SP-C used as a basis for the implementation and a construction of the implementation. Such constructions are represented by appropriate enrichments and/or renamings performed on top of SP-C. An implementation step is called *correct* if the application of the implementation construction to SP-C yields a behavioural implementation of SP-A.
In order to prove the correctness of an implementation step the ISAR system first normalizes all specification expressions. Then the *context induction prover*, the kernel of the system, is called for proving the implementation relation for the normalized specifications. Thereby all contexts and all proof obligations to be considered for the implementation proof are automatically generated. For the proof of equations the system is connected to the TIP system which is a narrowing-based inductive theorem prover (cf. [Fraus, Hußmann 91]). All steps of an implementation proof can be guided by appropriate interaction with the user. In particular, as usual when dealing with induction proofs, it is often necessary to find an appropriate generalization of the actual induction assertion if a nesting of context induction (implemented by a recursive call of the context induction prover) is performed. For that purpose the ISAR system generates automatically a set of particular contexts each context representing a generalization of the actual induction assertion. Then the user may select an appropriate context representing an assertion which is general enough for achieving successful termination of the proof algorithm.
As we will show by an example for the construction of generalized induction assertions it may be necessary to define additional function symbols which generalize (some) functions of the abstract specification. (For instance for the proof of the array pointer implementation of stacks a generalization of the *pop* operation by an operation *iterated_pop: nat, stack* $\rightarrow$ *stack* is used where *iterated_pop(n, s)* performs *n* pop

operations on a stack *s*.) Such function generalizations can be added as "hints" to an implementation step. Hints cannot be generated automatically. In this case the intuition of the system user is needed.

2 Basic Concepts

In this section we summarize the theoretical foundations of the ISAR system. In particular the notions of behavioural specifications and behavioural implementations are defined. Most definitions and results of this section can be found in [Hennicker 90] or (slightly revised) in [Hennicker 92].

2.1 Algebraic Preliminaries

First, we briefly review the basic notions of algebraic specifications which will be used in the following (for more details see e.g. [Ehrig, Mahr 85]). A (many sorted) *signature* Σ is a pair (S, F) where S is a set of *sorts* and F is a set of *function symbols* (also called *functions* for short). To every function symbol $f \in F$ a functionality $s_1,\ldots, s_n \to s$ with $s_1,\ldots, s_n \in S$ is associated. If $n = 0$ then f is called *constant* of sort s. A *signature morphism* $\rho: \Sigma \to \Sigma'$ between two signatures $\Sigma = (S, F)$ and $\Sigma' = (S', F')$ is a pair $(\rho_{sorts}, \rho_{functs})$ of mappings $\rho_{sorts}: S \to S'$, $\rho_{functs}: F \to F'$ such that for all $f \in F$ with functionality $s_1, \ldots, s_n \to s$, $\rho_{functs}(f)$ has functionality $\rho_{sorts}(s_1), \ldots, \rho_{sorts}(s_n) \to \rho_{sorts}(s)$. A signature $\Sigma' = (S', F')$ is called *subsignature* of Σ (written $\Sigma' \subseteq \Sigma$) if $S' \subseteq S$ and $F' \subseteq F$.
The *term algebra* $W_\Sigma(X)$ of all *Σ-terms* with variables of X (where $X = (X_s)_{s\in S}$ is an S-sorted family of sets of variables) is defined as usual. In particular, for any sort $s \in S$, $W_\Sigma(X)_s$ denotes the set of *terms* of sort s. If $X = \emptyset$ then $W_\Sigma(\emptyset)$ is abbreviated by W_Σ and W_Σ is called *ground term algebra.* We assume that any signature $\Sigma = (S, F)$ is *inhabited*, i.e. for each sort $s \in S$ there exists a ground term $t \in (W_\Sigma)_s$. A *substitution* $\sigma: X \to W_\Sigma(X)$ is a family of mappings $(\sigma_s: X_s \to W_\Sigma(X)_s)_{s\in S}$. For any term $t \in W_\Sigma(X)$, the *instantiation* $\sigma(t) =_{def} t[\sigma(x_1)/x_1, \ldots, \sigma(x_n)/x_n]$ is defined by replacing all variables $x_1, \ldots, x_n$ occurring in t by the terms $\sigma(x_1), \ldots, \sigma(x_n)$. A substitution $\sigma: X \to W_\Sigma$ is called *ground substitution.*

2.2 Behavioural Specifications

The syntax of behavioural specifications is defined similarly to [Nivela, Orejas 88] and [Reichel 85] where a distinguished set of sorts of a specification is declared as observable:
A *behavioural specification* SP = (Σ, Obs, E) consists of a signature $\Sigma = (S, F)$, a subset $Obs \subseteq S$ of *observable sorts* and a set E of *axioms*. Any behavioural specification is assumed to contain the observable sort *bool*, two constants *true, false:* $\to$ *bool* (denoting the truth values) and the axiom *true* $\neq$ *false*. The axioms of $E\backslash\{true \neq false\}$ are equations $t = r$ with terms $t, r \in W_\Sigma(X)$.

Specifications can be reused for the definition of new behavioural specifications by the operators *enrich* for enriching a given specification by some sorts, functions and axioms, + for the combination of two specifications and *rename* for renaming the sorts and functions of a specification. More precisely we define for any behavioural specification SP = (Σ, Obs, E) with signature Σ = (S, F):

enrich SP **by sorts** S1 **observable sorts** Obs1 **functions** F1 **axioms** E1 $=_{def}$
((S ∪ S1, F ∪ F1), Obs ∪ Obs1, E ∪ E1),

SP + SP1 $=_{def}$ (Σ ∪ Σ1, Obs ∪ Obs1, E ∪ E1) where SP1 = (Σ1, Obs1, E1),

rename SP **by** ρ $=_{def}$ (Σ1, ρ_{sorts}(Obs), ρ_{ax}(E))
where ρ: Σ → Σ1 is a bijective signature morphism and ρ_{ax} is the extension of ρ to Σ-formulas.

Note that the enrich operator is only defined if (S ∪ S1, F ∪ F1) forms a signature, Obs1 is a subset of S ∪ S1 and E1 are axioms over the signature (S ∪ S1, F ∪ F1). Moreover, note that in contrast to specification building operators in the sense of ASL (cf. [Wirsing 86]) the above operators are only defined syntactically in order to express textual abbreviations.

As an example, the following behavioural specification STACK describes the usual data structure of stacks with a constant *empty*, denoting the empty stack, an operation *push* for adding an element to a stack, an operation *top* for selecting the top element of a stack and an operation *pop* for removing the top element. STACK is an enrichment of BOOL and of an arbitrary specification ELEM of the elements of a stack. The sort *elem* for the elements is declared as observable while the sort *stack* is not observable. Hence, the behaviour of stacks can only be observed via their top elements.

```
spec STACK = enrich BOOL + ELEM by
   sorts stack
   observable sorts elem
   functions  empty: → stack,
              push: elem, stack → stack,
              top: stack → elem,
              pop: stack → stack
   axioms  top(push(e, s)) = e,
           pop(push(e, s)) = s   endspec
```

2.3 Behavioural Implementations

The definition of the behavioural implementation concept is based on the assumption that from the software user's point of view a software product is a correct implementation if it satisfies the desired input/output behaviour. Hence a behavioural specification SP1 = (Σ1, Obs1, E1) is called behavioural implementation of SP = (Σ, Obs, E) if SP1 respects the observable properties of SP. A precise formal definition of this intuitive notion using behavioural semantics (cf. e.g. [Nivela, Orejas 88], [Reichel 85]) is given in [Hennicker 90, 92]. Since we are interested in automated implementation proofs we will present here only the following proof theoretic characterization of behavioural implementations (cf. [Hennicker 90, 92]) which is the theoretical background of the ISAR system. The characterization uses the notion of a *Σ-context* which is any term $c[z_s]$ over the signature Σ of SP which contains a distinguished variable z_s of some sort $s \in S$ (where z_s occurs exactly once in $c[z_s]$). If the (result) sort, say s_0, of $c[z_s]$ belongs to Obs then $c[z_s]$ is called *observable Σ-context*. The application of a context $c[z_s]$ to a term t of sort s is defined by the substitution of z_s by t. Instead of $c[t/z_s]$ we also write briefly c[t]. In particular, for any sort s, the variable z_s is itself a Σ-context (called *trivial context*) of sort s and $z_s[t] = t$.

2.1 Proposition Let SP1 = (Σ1, Obs1, E1) and SP = (Σ, Obs, E) be behavioural specifications such that $\Sigma \subseteq \Sigma1$ and $Obs \subseteq Obs1$.
SP1 is a behavioural implementation of SP, if and only if for all observable Σ-contexts $c[z_s]$ and for all axioms $(t = r) \in E$ (such that t and r are of sort s),
$SP1 \vdash \sigma(c[t]) = \sigma(c[r])$ holds for all ground substitutions $\sigma: X \to W_\Sigma$.

In the above proposition $SP1 \vdash \sigma(c[t]) = \sigma(c[r])$ means that the equation $\sigma(c[t]) = \sigma(c[r])$ is deducible from the axioms of the implementation SP1 by the usual axioms and rules of the equational calculus, cf. e.g. [Ehrig, Mahr 85]. (The additional derivation rule (R) of [Hennicker 92] is only relevant for the necessity of the implementation condition if the implementation is inconsistent.) Since behavioural semantics in [Hennicker 90, 92] is restricted to term generated algebras it is enough to consider all ground instantiations $\sigma(c[t]) = \sigma(c[r])$ of the equations $c[t] = c[r]$.
Note that the "non observable" axioms t = r of an abstract specification SP need not to be satisfied by an implementation. Only the observable consequences of those axioms (formally expressed by applications of observable contexts) have to be satisfied by an implementation (for all ground substitutions). For instance, an implementation of the specification STACK not necessarily has to satisfy the non observable stack equation *pop(push(e, s)) = s* but it has to satisfy all (ground instantiations of) applications of observable contexts to this equation as e.g. *σ(top(pop(push(e, s)))) = σ(top(s))* with $\sigma: X \to W_\Sigma$.

3 The ISAR System

3.1 Implementation Proofs by Context Induction

Proposition 2.1 provides the starting point for an automatization of implementation proofs since it gives a proof theoretic characterization of behavioural implementations

where certain equations have to be derived from the axioms of the implementation. In particular, the proposition says that it is sufficient to show that all ground instantiations $\sigma(c[t]) = \sigma(c[r])$ of the equations $c[t] = c[r]$ are valid in the implementation SP1. Hence it is enough to prove that the equations $c[t] = c[r]$ are *inductive theorems* of SP1 where an equation e is called inductive theorem of SP1 if all ground instantiations of e are theorems of SP1, i.e. SP1 $\vdash \sigma(e)$ for all ground substitutions $\sigma: X \to W_{\Sigma 1}$ (cf. [Padawitz 88]). (Note that the inductive theorem property is slightly stronger than the condition of Proposition 2.1 since there only ground substitutions σ w.r.t. the subsignature $\Sigma \subseteq \Sigma 1$ are considered.) Then, for the proof of inductive theorems one may use theorem provers like the prover of Boyer and Moore (cf. [Boyer, Moore 88]) or the Larch Prover (cf. [Garland, Guttag 89]).
However, things are not that easy because, in general, infinitely many observable contexts exist and therefore one has to prove usually infinitely many equations $c[t] = c[r]$. Hence, for proving that SP1 is a behavioural implementation of SP, it is enough to show that the following property $P(c[z_s])$ is valid for all observable Σ-contexts $c[z_s]$:

$P(c[z_s]) = \text{true} \Leftrightarrow_{def}$ for all axioms $(t = r) \in E$ (such that t, r are of sort s), the equation $c[t] = c[r]$ is an inductive theorem of SP1.

Since observable Σ-contexts are particular terms (over the signature of the abstract specification) the syntactic subterm ordering defines a Noetherian relation on the set of observable Σ-contexts and therefore we can apply *context induction* (which is a particular instance of structural induction, cf. [Burstall 69]) for showing the validity of $P(c[z_s])$ for all observable Σ-contexts $c[z_s]$.
It turns out that in many cases implementation proofs by context induction work quite schematically although usually a lot of different cases of contexts have to be distinguished. Hence it is the aim of the ISAR system to provide a tool which automates (to a certain extent) implementation proofs by context induction. The principle idea for executing implementation proofs by the ISAR system is the following one: (For a detailed description of the underlying algorithm of the system we refer to [Hennicker 92].)
Let SP and SP1 be as above. In the first step (which is the base of the context induction) it has to be shown that $P(z_s)$ is valid for all trivial observable Σ-contexts z_s (which just are variables of observable sort s). According to the definition of the property P this means that one has to prove that all "observable" axioms $t = r$ of SP (with terms t and r of observable sort) are inductive theorems of SP1. For the proof of the equations the ISAR system is connected to the TIP system (cf. [Fraus, Hußmann 91]) which is a narrowing-based inductive theorem prover.
In the second step, the induction step is performed for all contexts of the form $f(\ldots,c[z_s],\ldots)$ where f is a function symbol of SP with observable result sort and $c[z_s]$

ranges over all Σ-contexts of sort, say s_i. Then, for the proof of the actual induction assertion it has to be shown that for all Σ-contexts $c[z_s]$ of sort s_i, $P(f(\ldots,c[z_s],\ldots))$ holds, i.e. for all axioms $t = r$ of SP (such that t, r are of sort s) the equation $f(\ldots,c[t],\ldots) = f(\ldots,c[r],\ldots)$ is an inductive theorem of SP1. For that purpose two cases are distinguished:

Case 1: s_i is an observable sort of SP. Then, by hypothesis of the context induction, $P(c[z_s])$ holds, i.e. $c[t] = c[r]$ is an inductive theorem of SP1 for all axioms $t = r$ of SP. Hence $f(\ldots,c[t],\ldots) = f(\ldots,c[r],\ldots)$ is also an inductive theorem of SP1 for all axioms $t = r$ of SP, i.e. $P(f(\ldots,c[z_s],\ldots))$ holds.
Case 2: s_i is not an observable sort of SP. Then, the hypothesis of the context induction cannot be applied for $c[z_s]$ and therefore a nested context induction (over all Σ-contexts $c[z_s]$ of sort s_i) is performed for proving the property $P(f(\ldots,c[z_s],\ldots))$ for all Σ-contexts $c[z_s]$ of sort s_i.

In the ISAR system each nesting of context induction is implemented by a recursive call of the *context induction prover* where the actual parameter is a (fixed) context $c0[z_{s_i}]$ which represents the actual induction assertion "$P(c0[c[z_s]])$ is valid for all Σ-contexts $c[z_s]$ of sort s_i". (For instance, the induction assertion "$P(f(\ldots,c[z_s],\ldots))$ is valid for all Σ-contexts $c[z_s]$ of sort s_i" is represented by the context $f(\ldots,z_{s_i},\ldots)$.) Initially the context induction prover is called for all trivial contexts z_{s_0} of observable sort $s_0 \in S$. Then, if all steps of the proof procedure are performed the principle of context induction implies that the property $P(c[z_s])$ is proved for all observable Σ-contexts $c[z_s]$ and hence it is shown that SP1 is a behavioural implementation of SP. Obviously, the proof algorithm of the ISAR system cannot be complete (in the sense that all valid implementation relations can be proved by ISAR) because context induction and inductive theorem proving are not complete.
In order to achieve successful termination of the implementation proof it is often necessary to find an appropriate generalization of the actual induction assertion. Therefore automated reasoning has to be supplemented by interaction with the user who may select before each nesting of context induction a context which represents a generalization of the actual induction assertion. For instance, any subcontext $c0'[z_{s_i}]$ of a context $c0[z_{s_i}]$ represents a generalization of the assertion represented by $c0[z_{s_i}]$ because it is easy to show (using the congruence rule of the equational calculus) that in this case $P(c0'[c[z_s]])$ implies $P(c0[c[z_s]])$ for all Σ-contexts $c[z_s]$ of sort s_i. Hence, instead of $c0[z_{s_i}]$ any subcontext $c0'[z_{s_i}]$ can be correctly used as an actual parameter of a recursive call of the proof algorithm.
The ISAR system generates automatically before each nesting of context induction all subcontexts of the context $c0[z_{s_i}]$ which represents the actual induction assertion and the user may choose an appropriate one. Besides the generalization of contexts by subcontexts there are two further constructions of context generalizations performed by the ISAR system: The first one allows to abstract from a context $c0[z_s]$ by

replacing subterm(s) of $c0[z_S]$ (which do not contain z_S) by variables. The second one allows to construct new contexts by applying rewrite steps to the original context $c0[z_S]$. An example for the generation of context generalizations can be seen in the implementation proof of Section 3.3.

3.2 Implementation Steps

Usually the implementation of an abstract specification is constructed on top of already existing specifications of concrete data structures like lists, arrays, trees etc. (see e.g. [Ehrig et al. 82], [Sannella, Tarlecki 88] for formalizations of implementation constructions). In order to document the construction of the implementation the input of the ISAR system is the description of an *implementation step* consisting of three parts: an abstract specification SP = (Σ, Obs, E) to be implemented, `a concrete specification SP-C = (Σ-C, Obs-C, E-C) used as a basis for the implementation and a construction of the implementation on top of SP-C. The implementation construction can be defined by some enrichment and/or renaming of SP-C. For instance, the following implementation step performs first a renaming of SP-C w.r.t. a signature morphism ρ and then an enrichment Δ = **sorts** S1 **observable sorts** Obs1 **functions** F1 **axioms** E1 of the renamed version of SP-C:

```
implementation step SP_by_SP-C =
     SP is implemented by SP-C
     via renaming ρ, enrichment Δ
endimplstep
```

Such an implementation step is called *correct* if

enrich (**rename** SP-C **by** ρ) **by** Δ is a behavioural implementation of SP.

The correctness of implementation steps when performing first an enrichment and then a renaming is defined analogously. Before starting an implementation proof the *normalizer* of the ISAR system computes normal forms of all specifications according to the definition of the operators *enrich*, *rename* and + (cf. Section 2.2).

In some implementation proofs it may be necessary to use particular lemmas (i.e. theorems which are valid in the implementing specification) and even auxiliary function definitions which are used for the construction of contexts which represent sufficiently general induction assertions. Such lemmas and function definitions can be added as "hints" to an implementation step. Hints cannot be generated automatically. In this case the intuition of the system user is necessary. However, it seems that in most examples the auxiliary functions can be created just by generalizations of those abstract functions which would be iteratively used in recursive calls of the proof procedure (cf. the generalization of the *pop* operation by the operation *iterated_pop* below.)

As an example we consider an implementation step which implements stacks on top of the following specification of (dynamic) arrays. For simplicity only those array operations are defined which are necessary for the example: *vac* denotes the empty array, *put* inserts an element into an array at a particular index and *get* delivers the actual value for a given index. The indices are natural numbers which are specified in the underlying specification NAT. It is assumed that the specification ELEM of the array elements contains a constant *constelem* and a conditional function *ifelem . then . else . fi: bool, elem, elem → elem.*

```
spec ARRAY=
   enrich BOOL + NAT + ELEM by
     sorts array
     observable sorts elem
     functions vac  : -> array,
               put  : array, nat, elem -> array,
               get  : nat, array -> elem
     axioms
        get(k, put(a, l, e)) =
             ifelem eq_nat(k, l) then e else get(k, a) fi,
        get(k, vac) = constelem   endspec
```

Then the implementation of stacks on top of arrays is defined by the following implementation step:

```
implementation step STACK_by_ARRAY =
     STACK is implemented by ARRAY
     via  enrichment
            sorts stack
            functions  pair : array, nat -> stack,
                       empty : -> stack,
                       push : elem, stack -> stack,
                       top : stack -> elem,
                       pop : stack -> stack
            axioms
                empty = pair(vac, 0),
                push(e, pair(a, p)) =
                     pair(put(a, p+1, e), p +1),
                top(pair(a, p)) = get(p, a),
                pop(pair(a, p)) = pair(a, p-1)
   hints
      auxiliary functions
            iterated_pop : nat, stack -> stack
      axioms
            iterated_pop(zero, s) = s,
            iterated_pop(n+1, s) = iterated_pop(n, pop(s))
      lemmas
            iterated_pop(i, pair(a,n)) = pair(a, n-i)
endimplstep
```

In the above implementation construction stacks are implemented by their familiar array pointer representation, i.e. by pairs consisting of an array and a pointer (a natural number) which points to the top element of a stack. The stack operations *empty, push, top* and *pop* are implemented as usual. For instance, the *pop* operation simply decrements the pointer without deleting the entry at the last top position. Hence the abstract stack equation *pop(push(e, s)) = e* is not valid in the implementation. But we will see that nevertheless the implementation step is correct since the implementation satisfies all observable consequences of the abstract stack equations. The usefulness and necessity of the hints will be seen in the following when the implementation proof is performed by the system.

3.3 An Example Session with the ISAR System

After the ISAR system is called we first give a command for reading the file where the implementation step STACK_by_ARRAY together with the specifications STACK and ARRAY is stored. After syntactical analysis all specifications and the implementation step are normalized and it is checked whether the signature and the observable sorts of the abstract specification are included in the (normalized) implementation because this is the precondition of our implementation definition. It is possible to display all specifications and the implementation step in their normal form and in their structured form using the commands `list norm` or `list struct`. Now the implementation proof can be started by calling the context induction prover. In the following we will show how the system performs the implementation proof and we will give detailed comments - written in *italics* - which do not belong to the output of the system.

```
****** CONTEXT-INDUCTION ******

Observable Sorts of STACK:
bool, elem

***** PROOF-OBLIGATION: ******
FOR ALL CONTEXTS c OF SORT bool, elem AND FOR ALL AXIOMS t=r OF
STACK IT IS TO SHOW:
IMPLEMENTATION |- c[t] = c[r]
```

(This is exactly the property P(c) defined in Section 3.1 (since induction is allowed for the derivation). In the next step the axioms of STACK are considered explicitely. Thereby the axioms of the underlying specifications BOOL and ELEM are omitted because these axioms belong already to the implementation. The variables X29, X30, ... are system variables.)

```
THE FOLLOWING EQUATIONS HAVE TO BE PROVED
FOR ALL CONTEXTS c OF SORT bool, elem:
  c[top(push(X29, X30))] = c[X29]   (if the context variable z_s
                                     has sort s = elem)
  c[pop(push(X31, X32))] = c[X32]   (if the context variable z_s
                                     has sort s = stack)
```

```
* BASE OF THE CONTEXT INDUCTION FOR CONTEXTS OF SORT: bool, elem
* DEPTH: 0
  ONLY THE OBSERVABLE AXIOMS HAVE TO BE PROVED
```

(For the base of the context induction the property $P(z_{elem})$ has to be proved where z_{elem} is the trivial context of sort elem. Note that $P(z_{bool})$ is trivially satisfied since no axiom of sort bool is considered.)

```
* PROOF OBLIGATIONS:
  (1): top(push(X29, X30)) = X29

ISAR/PROOF>all
```

(The command "all" says that all equations of the proof obligation - here only one exists - are selected to be proved. For the proof of the equation the ISAR system calls the TIP system.)

```
  top(push(X29, X30)) = X29
# EQUATION PROVED BY TIP-SYSTEM.
```

(The proof of this equation is performed by induction on the structure of X30 where the function "pair" is used as a constructor for stacks. In general, constructors can be chosen interactively during the TIP proof or they can be declared previously in the implementing specification. Constructor completeness can be checked by the TIP system and a listing of the TIP proof can be displayed if required.)

```
* BASE OF THE CONTEXT INDUCTION OF DEPTH 0 FINISHED

*CONTEXT INDUCTION STEP FOR CONTEXTS OF SORT: bool, elem
* DEPTH: 0
```

(It is enough to perform the induction step for contexts with outermost function symbol top. For all other function symbols of STACK with result bool or elem the argument sorts are observable as well and hence the induction step is trivial.)

```
* THE FOLLOWING ABSTRACT FUNCTION WITH OBSERVABLE RESULT-SORT
* HAS TO BE CONSIDERED
  (1): top : stack -> elem

ISAR/PROOF>all
  Selected function: top : stack -> elem
```

(At this point the actual induction assertion to be proved by a nested context induction is: "P(top(c)) is valid for all contexts c of sort stack". However in the induction step of the nested context induction contexts of the form top(pop(c)) have to be considered but it is not possible to prove P(top(pop(c))) using the hypothesis P(top(c)). Hence one could try to start a second nesting of context induction for proving P(top(pop(c))). But then the same situation occurs leading to a further iteration of context induction for proving P(top(pop(pop(c)))) etc. Consequently, the implementation proof would not terminate. A possible solution is to code the iteration of the pop operation by an auxiliary function iterated_pop: nat, stack → stack such that iterated_pop(n, s) pops a stack s n-times. The operation iterated_pop is specified in the hints of the implementation step.
Now having the hints available the ISAR system generates the following set of contexts. Context (1) represents the original induction assertion which was not successful. Context (2) represent a more general induction assertion (namely: "P(c) is valid for all contexts c of sort stack") but since the non observable STACK axiom pop(push(e, s)) = s is not valid in the implementation this assertion is too general. Context (3) is

equivalent to context (1) by the first axiom of the hints. Context (4) is a generalization of context (3) where the term zero is replaced by a variable. We will see that the selection of (4) will be successful.)

```
* GENERATED CONTEXTS:
  (1): top(z_stack)
  (2): z_stack
  (3): top(iterated_pop(zero, z_stack))
  (4): top(iterated_pop(X145, z_stack))

ISAR/PROOF>sel 4
```

(Now the assertion represented by context (4) is displayed.)

```
***** PROOF-OBLIGATION: ******
FOR ALL CONTEXTS c OF SORT stack AND FOR ALL AXIOMS t=r OF STACK
IT IS TO SHOW:
IMPLEMENTATION |- top(iterated_pop(X145,c[t])) =
                  top(iterated_pop(X145,c[r]))

THE FOLLOWING EQUATIONS HAVE TO BE PROVED
FOR ALL CONTEXTS c OF SORT stack:
  top(iterated_pop(X145, c[pop(push(X31, X32))])) =
  top(iterated_pop(X145, c[X32]))

* BASE OF THE CONTEXT INDUCTION FOR CONTEXTS OF SORT: stack
* DEPTH: 1
```

(Depth 1 indicates the depth of the nesting of context induction which is actually performed.)

```
* PROOF OBLIGATIONS:
  (1): top(iterated_pop(X145, pop(push(X31, X32)))) =
       top(iterated_pop(X145, X32))

ISAR/PROOF>all

  top(iterated_pop(X145, pop(push(X31, X32)))) =
  top(iterated_pop(X145, X32))
# EQUATION PROVED BY TIP-SYSTEM.
```

(For the proof of this equation the lemma stated as a hint in the implementation step is used. The proof is again performed by induction on the structure of X32 using the constructor "pair".)

```
* BASE OF THE CONTEXT INDUCTION OF DEPTH 1 FINISHED

* CONTEXT INDUCTION STEP FOR CONTEXTS OF SORT: stack
* DEPTH 1

* THE FOLLOWING ABSTRACT FUNCTIONS WITH RESULT-SORT stack
* HAVE TO BE CONSIDERED
  (1): push : elem, stack -> stack
  (2): pop : stack -> stack

ISAR/PROOF>all
  Selected function: push : elem, stack -> stack
* WHICH ARGUMENT?
ISAR/PROOF>all
  push : elem, stack -> stack
```

(Here the system does not select the first argument sort of push since elem is an observable sort and hence the induction step is trivial. For the second argument sort the actual induction assertion to be proved is: "P(top(iterated_pop(X145, push(X729, c)))) is valid for all contexts c of sort stack." Thereby the induction hypothesis of the nested context induction can be

```
applied, i.e. one has to show that for all contexts c of sort
stack, P(top(iterated_pop(X145,c))) implies P(top(iterated_pop
(X145, push(X729, c)))). For this it is enough to show for newly
introduced constants constant1_stack and constant2_stack of sort
stack that the proof obligation below can be derived from the
implementation if the following additional hypothesis is added
to the axioms of the implementation:)

* ADDITIONAL HYPOTHESIS OF THE CONTEXT INDUCTION:
  top(iterated_pop(X145, constant1_stack)) =
  top(iterated_pop(X145, constant2_stack))
* PROOF OBLIGATION:
  top(iterated_pop(X145, push(X729, constant1_stack))) =
  top(iterated_pop(X145, push(X729, constant2_stack)))
# EQUATION PROVED BY TIP-SYSTEM.

(The proof is performed by induction over X145 using the
constructors "zero" and "succ" of the natural numbers. The base
of the induction uses the equation top(push(X29, X30)) = X29
which has been proved previously and therefore is automatically
added to the lemmas of the implementation. The induction step
uses the equation top(iterated_pop(X145, pop(push(X31, X32))))
= top(iterated_pop(X145, X32)) (which has also been proved
before) and the hypothesis of the context induction.)

* THE FOLLOWING ABSTRACT FUNCTIONS WITH RESULT-SORT stack
* HAVE TO BE CONSIDERED
  (1): push : elem, stack -> stack * proved
  (2): pop : stack -> stack

  Selected function: pop : stack -> stack

* ADDITIONAL HYPOTHESIS OF THE CONTEXTT INDUCTION:
  top(iterated_pop(X145, constant1_stack)) =
  top(iterated_pop(X145, constant2_stack))
* PROOF OBLIGATION:
  top(iterated_pop(X145, pop(constant1_stack))) =
  top(iterated_pop(X145, pop(constant2_stack)))
# EQUATION PROVED BY TIP-SYSTEM.

(The proof uses the second axiom of iterated_pop and the
hypothesis.)

* CONTEXT INDUCTION OF DEPTH 1 FINISHED

* CONTEXT INDUCTION OF DEPTH 0 FINISHED

* END OF THE IMPLEMENTATION PROOF
***** ALL PROOF OBLIGATIONS PROVED! *****
```

3.4 The Structure of the ISAR System

The ISAR system is written in the programming language PASCAL in order to be compatible with the TIP system which verifies all proof obligations generated by ISAR. The main modules of ISAR are

- a *scanner* and a *parser* with mixfix-parser for the syntactical analysis of specifications and implementation steps,
- a *normalizer* for flattening structured specifications and implementation steps,
- a *context generator* to produce automatically contexts which represent generalizations of the actual induction assertion,

- a *TIP-interface* for the exchange of informations between the ISAR and the TIP system and
- the proof-modules of the *TIP-system* for the verification of the proof obligations.

For lack of space we cannot give here a more elaborated description of the internal structure and technical details of the ISAR system. Interested readers may consult [Bauer 93].

4 Concluding Remarks

The development of the ISAR system is the consequent third step after having introduced the context induction principle for proving behavioural implementations in [Hennicker 90] and after the investigation of a proof procedure for context induction proofs in [Hennicker 92]. The proof techniques of the ISAR system and the underlying implementation concept are based on behavioural semantics which is a major difference for instance to the ISDV system (cf. [Beierle, Voß 85]) where implementations and abstract specifications are related by representation homomorphisms. Recently, in [Bidoit, Hennicker 92] a method was developed for proving observational theorems over a behavioural specification with the Larch Prover (cf. [Garland, Guttag 88]) where (under particular assumptions) an explicit use of context induction could be avoided by using the partioned by deduction rule of LP. It is an interesting objective of future research how an environment for proving behavioural implementations could be built on top of LP.
The actual version of the ISAR system is restricted to equational specifications but it is intended to provide an extension to conditional equational specifications (with observable premises of the axioms) and to implementations of parameterized specifications. In particular, for dealing with parameterized implementations the solution is very simple: One just has to guarantee that proofs of equations by the TIP system do not use induction on parameter sorts.
An important direction of future development concerns the use of ISAR for the representation of reusable software components and for retrieving reusable components from a component library. In fact an implementation step as it is processed by ISAR represents a two level reusable component in the sense of [Wirsing 88] where the specification to be implemented represents the abstract description of the component's behaviour and the implementation represents the realisation of the component. (Actual research deals with an extension of the implementation notion such that object-oriented classes with imperative method definitions can be used as implementations.) Concerning the retrieval of components the ISAR system provides already a tool which allows to check whether the reuse of components which are retrieved from a component library by syntactic signature matching (cf. [Chen et al. 93]) is semantically correct with respect to a given goal specification. As a consequence, we suggest to combine both techniques, syntactic signature matching and correctness proofs by ISAR, in order to obtain a complete retrieval system for reusable software components.

Acknowledgement We would like to thank Martin Wirsing for several valuable suggestions for the design of the ISAR system. This work is partially sponsored by the German BMFT project KORSO.

References

[Bauer 93] An interactive system for algebraic implementation proofs: The ISAR system from the user's point of view. Universität München, Technical Report (to appear), 1993.

[Beierle, Voß 85] C. Beierle, A. Voß: Algebraic specification and implementation in an integrated software development and verification system. MEMO SEKI-12, FB Informatik, Universität Kaiserslautern, 1985.

[Bidoit, Hennicker 92] How to prove observational theorems with LP. Proc. of the First International Workshop on Larch, July 1992, Boston, USA, Springer Verlag Workshop in Computing Series, 1993. Also in: Laboratoire d'Informatique de l' Ecole Normale Supérieure, Paris, LIENS-92-23, 1992.

[Boyer, Moore 88] R. S. Boyer, J. S. Moore: A computational logic handbook. Academic Press, New York, 1988.

[Burstall 69] R. M. Burstall: Proving properties of programs by structural induction. *Comp. Journal* **12**, 41-48, 1969.

[Chen et al. 93] P. S. Chen, R. Hennicker, M. Jarke: On the retrieval of reusable software components. In: R. Prieto-Diaz, W. B. Frakes (eds.): Advances in Software Reuse. *Selected Papers from the Second International Workshop on Software Reusability.* Lucca, Italy, 1993. IEEE Computer Society Press, Los Alamitos, California, Order Number 3130, 99-108, 1993.

[Ehrig et al. 82] H. Ehrig, H.-J. Kreowski, B. Mahr, P. Padawitz: Algebraic Imple-mentation of Abstract Data Types. *Theoretical Computer Science* **20**, 209-263, 1982.

[Ehrig, Mahr 85] H. Ehrig, B. Mahr: Fundamentals of algebraic specification 1, EATCS Monographs on Theoretical Computer Science **6**, Springer, Berlin, 1985.

[Fraus, Hußmann 91] U. Fraus, H. Hußmann: A narrowing-based theorem prover. Extended Abstract. In: Proc. RTA '91, Rewriting Techniques and its Applications, Lecture Notes in Computer Science **488**, 435-436, 1991.

[Garland, Guttag 88] S. J. Garland, J. V. Guttag: An overview of LP, the Larch Prover. In: Proc. RTA '89, Rewriting Techniques and its Applications, Lecture Notes in Computer Science **355**, 137-151, 1989.

[Hennicker 90] R. Hennicker: Context Induction: a proof principle for behavioural abstractions. In: A. Miola (ed.): Proc. DISCO '90, International Symposium on Design and Implementation of Symbolic Computation Systems, Capri, April 1990. Lecture Notes in Computer Science **429**, 101-110, 1990.

[Hennicker 92] A semi-algorithm for algebraic implementation proofs. *Theoretical Computer Science* **104**, Special Issue, 53-87, 1992.

[Nivela, Orejas 88] Mª P. Nivela, F. Orejas: Initial behaviour semantics for algebraic specifications. In: D. T. Sannella, A. Tarlecki (eds.): Proc. 5th Workshop on Algebraic Specifications of Abstract Data Types, Lecture Notes in Computer Science **332**, 184-207, 1988.

[Padawitz 88] P. Padawitz: Computing in Horn clause theories. EATCS Monographs on Theoretical Computer Science **16**, Springer, Berlin, 1988.

[Reichel 85] H. Reichel: Initial restrictions of behaviour. IFIP *Working Conference*, The Role of Abstract Models in Information Processing, 1985.

[Sannella, Tarlecki 88] D. T. Sannella, A. Tarlecki: Toward formal development of programs from algebraic specifications: implementation revisited. *Acta Informatica* **25**, 233-281, 1988.

[Wirsing 86] M. Wirsing: Structured algebraic specifications: a kernel language. *Theoretical Computer Science* **42**, 123-249, 1986.

[Wirsing 88] Algebraic description of reusable software components. In: Proc. COMPEURO '88, Comp. Society Order Number 834, 300-312, 1988.