Equational Specifications: Design, Implementation, and Reasoning*

S. Antoy[1], P. Forcheri[2], J. Gannon[3], and M. T. Molfino[2]

[1]*Department of Computer Science*
Portland State University
Portland, Oregon.

[2]*Istituto per la Matematica Applicata*
Consiglio Nazionale delle Ricerche
Genova, Italy.

[3]*Department of Computer Science*
University of Maryland
College Park, Maryland.

Abstract

Sets of equations can be used to specify, implement, and reason about software. We discuss how to automate these tasks for constructor-based, convergent rewrite systems. Using incremental design strategies, we obtain completely defined, consistent, and sufficiently complete specifications. Direct implementations of specifications as term rewriting systems serve as software prototypes of systems. We use prototypes to determine that specifications are consistent with our intuitive expectations during design and with more efficient implementations during testing. We describe an automated tool for reasoning about both the properties a specification and the correctness of its implementation. Our approach is applicable to a relatively small class of specifications, but within this class it appears to be effective for designing high quality specifications and for effectively using these specifications for a variety of other tasks arising during the software lifecycle.

*This work has been partially supported by the C.N.R. "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" under grant n.90.00750.69, by the National Science Foundation grant CCR-9196023 and CCR-9406751, and by the Office of Naval Research grants N00014-87-K-0307 and N0014-90-J4091.

1 Introduction

Sets of equations specify software systems either by describing the result of a computation or by characterizing some properties of the result. Suppose that the problem at hand is that of sorting a sequence of elements. We specify an operation *sort*, using an auxiliary operation *insert*, as follows:

$$\begin{aligned}
&\mathit{sort}(\mathit{nil}) = \mathit{nil}\\
&\mathit{sort}(\mathit{cons}(E, L)) = \mathit{insert}(E, \mathit{sort}(L))\\
&\mathit{insert}(E, \mathit{nil}) = \mathit{cons}(E, \mathit{nil})\\
&\mathit{insert}(E, \mathit{cons}(F, L)) = \textbf{if } E \le F \textbf{ then } \mathit{cons}(E, \mathit{cons}(F, L)) \textbf{ else } \mathit{cons}(F, \mathit{insert}(E, L))
\end{aligned}$$

Here, we assume that sequences are constructed by *nil* and *cons* and that "$\le$" denotes some ordering relation among the elements of a sequence. These equations specify the result of applying the operation *sort* to a sequence of elements.

An alternative approach consists in specifying properties of the result of applying the operation *sort* to a sequence of elements. There are two relevant properties. One is that of "being *sorted*," which is specified as follows:

$$\begin{aligned}
&\mathit{sorted}(\mathit{nil}) = \mathit{true}\\
&\mathit{sorted}(\mathit{cons}(E, \mathit{nil})) = \mathit{true}\\
&\mathit{sorted}(\mathit{cons}(E, \mathit{cons}(F, L))) = E \le F \textbf{ and } \mathit{sorted}(\mathit{cons}(F, L))
\end{aligned}$$

The other property of the result is that of "being a *permutation*" of the input. Informally, the sequences L and M are permutations of each other if and only if any element occurs the same number of times in L and M.

Both specifications are concerned with describing the input/output relation of the same problem, thus they can be easily related to each other:

$$\mathit{sort}(X) = Y \quad \text{if and only if} \quad \mathit{sorted}(Y) \textbf{ and } \mathit{permutation}(X, Y)$$

Although each specification is complete for the problem at hand and independent of the other, we regard them as complementary, rather than alternative. Concordance of independent specifications alleviates a problem occurring in the early phases of the software lifecycle. Decisions made during later phases, such coding and testing, can be traced to decisions of earlier phases, for example, a formal specification. However, early decisions, such as the specification itself, are based on information and knowledge which is seldom formalized. The checkable redundancy of the two forms of specification may allow us to catch some errors or to increase our confidence that a problem is well understood.

In addition to the above advantages, obvious differences in the two specifications may allow us to attack a problem from different angles. For example, the first specification is simple and direct, but may implicitly suggest an implementation of the sorting technique known as straight insertion. The second specification is more declarative, but does not immediately guarantee either the existence or uniqueness of sorting, and makes its implementation less obvious.

This note is concerned with the design, validation, use, and integration of different forms of specification. In Section 2 we discuss the framework and the notation we use for our specifications. In Section 3 we present some techniques for designing specifications with desirable properties of completeness and consistency. In Section 4 we describe the direct implementation of our specifications and uses of this implementation. In Section 5 we outline the structure of an automated prover for our class of specifications and its use for code verification.

Some aspects of our discussion have been treated, perhaps more deeply, by others, for example, OBJ [16] and *Larch* [14]. Our approach emphasizes design and the dual role of specifications: description of behavior and description of properties. We address the integration of these roles and the application of the first to prototyping and testing, and the second to verification and reasoning.

2 Specification Framework

We use an abstract language to present specifications. Lower case identifiers are symbols of the signature. If a signature is constructor-based, its constructors will be explicitly mentioned. The arity and co-arity of a symbol will generally be inferred from the context, or will be explicitly mentioned when non-obvious. Variables, which are sorted, are denoted by upper case letters. The variables in an equation are universally quantified. We are mainly concerned with the design and use of axioms; thus our specifications often consist only of equations.

Given a signature $\mathcal{F}$ and a set E of equations, the meaning of our syntactic constructs is given by the initial $\mathcal{F}-$algebra among all models of E. Reasoning based on the replacement of equals for equals is clearly sound, and it is complete in the sense of the Birkhoff's Theorem.We use rewriting [12, 19] to restrict the freedom of replacing equals for equals, which is operationally very difficult to control. For convergent rewrite systems, equational reasoning is implemented by reduction to normal form without loss of reasoning power.

A traditional solution to employ rewrite rules instead of equations consists in transforming the equations of a specification into a corresponding rewrite system by means of the Knuth-Bendix completion procedure [21]. Although this approach appears simple to the specifier, the completion procedure may fail, may not terminate, or may require user intervention for ordering some rules it generates. We will discuss an alternative, based on design strategies, that yields a specification that is already in the form of a convergent rewrite system.

Our strategies require constructor-based signatures. This condition is not overly restrictive for the specification of software, but introduces the new problem of the sufficient completeness of the specification. We will address this problem, too. The signature of a specification is partitioned into constructors and (defined) operations when there exists a clear separation between data and computations. Every term built only from constructors represents a datum and every datum is represented by a constructor term. Every operation represents a computation and every computation is represented by a term containing an operation symbol. This viewpoint leads to considerable benefits discussed shortly. Referring to the examples of the introduction, *nil* and *cons* are the constructors of the type *sequence*, whereas *sort* and *sorted* are operations defined on the type *sequence*.

Equational reasoning may be too weak for proving all the interesting properties of a specification. Some properties may be provable via structural induction [9] or data type induction [17]. We employ the following principle. An inductive variable of type T is replaced by terms determined by T's constructors and inductive hypotheses are established. If F is a formula to be proved, v is the inductive variable, and s is the type of v, our induction proofs are carried out in the following manner. For every constructor c of type $s_1 \times \ldots \times s_n \rightarrow s$ for $n \geq 0$, we prove $F[c(v_1, \ldots, v_n)/v]$, where v_i, $1 \leq i \leq n$, is a distinct Skolem constant; and if $s_i = s$, then $F[v_i/v]$ is an inductive hypothesis.

3 Specification Design

3.1 Completeness and Consistency of Definition

A goal of any specification is to avoid the errors of saying too much, which may lead to inconsistency, or too little, which may lead to incompleteness. Signatures with constructors help us to avoid these errors. A constructor-based specification should describe the behavior of an operation on every combination of constructor terms of the appropriate type and only on these terms. Ideally, for every combination of arguments of an operation there is one and only one axiom that defines the behavior of the operation on those arguments. A technique called the *Binary Choice Strategy* (BCS) allows us to produce left sides of axioms with this property. The BCS is conveniently explained by means of an interactive, iterative, non-deterministic procedure that through a sequence of binary decisions generates the left sides of the axioms of a defined operation. We used the symbol "$\Box$," called *place*, as a placeholder for a decision. Let f be an operation of type $s_1, \ldots, s_k \rightarrow s$. Consider the template $f(\Box, \ldots, \Box)$, where the i^{th} place has type s_i. To construct the set of left sides of rules for f, we replace, one at a time, each place of a template with either a variable or with a series of constructor terms of the appropriate type. In forming the left sides, we neither want to forget some combination of arguments, nor include other combinations twice. That is, we want to avoid both underspecification and overspecification. This is equivalent to forming a *constructor enumeration* [10].

We achieve our goal by selecting a place in a template and choosing one of two options: "*variable*" or "*inductive*." The choice *variable* replaces the selected place with a fresh variable. The choice *inductive* for a place of type s_i splits the corresponding template in several new templates, one for each constructor c of type s_i. Each new template replaces the selected place with $c(\Box, \ldots, \Box)$, where there are as many places as the arity of c. A formal description of the strategy appears in [1]. The BCS produces the left sides of the rules of the operation *sorted* shown in the Introduction as follows. The initial template is:

$sorted(\Box)$

We choose *inductive* for the place. Since this place is of type *sequence*, we split the template into two new templates, one associated with *nil* and the other with *cons*:

$sorted(nil)$
$sorted(cons(\Box, \Box))$

We now choose *variable* for the first place of the second template and then *inductive* for the second place to obtain:

$sorted(nil)$
$sorted(cons(E, nil))$
$sorted(cons(E, cons(\Box, \Box)))$

We choose *variable* for last remaining places and obtain the left sides of the axioms shown in the Introduction.

If the constructors of the signature are free, the BCS produces axioms with non overlapping left sides and consequently avoids overspecification and inconsistencies in a specification. The BCS also produces completely defined operations. This, however, is not enough to avoid underspecification or to ensure that the specification is sufficiently complete. To achieve this other goal we need the strategy described next.

3.2 Sufficient Completeness and Termination

If a specification is terminating and its operations are completely defined, as provided by the BCS, then the specification is also sufficiently complete. We use a technique called *Recursive Reduction Strategy* (RRS) to guarantee the termination, and thus the sufficient completeness, of a specification. The RRS is conveniently explained by means of a function mapping terms to terms. The recursive reduction of a term t is the term obtained by "stripping" t of its recursive constructors. A constructor of type s is called *recursive* if it has some argument of type s. For example, *cons* is a recursive constructor of *sequence* because its second argument is of type *sequence*. Informally, "stripping" a term t is the operation of replacing any subterm of t rooted by a recursive constructor with its recursive argument. The stripping process is recursively applied throughout the term. A formal description of the recursive reduction function appears in [1]. We show its application in examples.

For reasons that will become clear shortly, we are interested in computing the recursive reduction of the left side of a rewrite rule for use in the corresponding right side. The symbol "\$" in the right side of a rule denotes the recursive reduction of the rule's left side. With this convention, the second axiom of the operation *insert* is written:

$$\mathit{insert}(E, \mathit{cons}(F, L)) = \textbf{if } E \leq F \textbf{ then } \mathit{cons}(E, \mathit{cons}(F, L)) \textbf{ else } \mathit{cons}(F, \$)$$

since the recursive reduction of the left side is $\mathit{insert}(E, L)$. We obtain it by replacing $\mathit{cons}(F, L)$ with L, since L is the recursive argument of *cons*. Likewise, the second axiom of *sort* is written:

$$\mathit{sort}(\mathit{cons}(E, L)) = \mathit{insert}(E, \$)$$

When a constructor has several recursive arguments the recursive reduction of a term may require an explicit indication of the selected argument. We may also specify a partial, rather than complete, "stripping" of the recursive constructors. This would be appropriate for the operation *sorted*. In the third axiom, the recursive reduction of $\mathit{sorted}(\mathit{cons}(E, \mathit{cons}(F, L)))$ is $\mathit{sorted}(L)$, but we strip only the outermost constructor and in the right side we use $\mathit{sorted}(\mathit{cons}(F, L))$.

It is not difficult to see that the recursive reduction of a term t containing recursive constructors yields a term smaller than t. This property is the key to ensuring termination. We design a specification incrementally as follows. Suppose that S_i is a "well-behaved" specification, i.e., completely defined, consistent, terminating, and sufficiently complete. We extend S_i with some new operations using the previously discussed strategies. We obtain a new specification, S_{i+1}, that is still well behaved and is a conservative extension, in the sense of [13], of S_i.

We show how our technique produces the first specification of the Introduction. We assume that the operation "$\leq$," generic in our specification, is defined within a well-behaved specification. The "**if-then-else**" operation, which we consider a primitive, is well behaved too. The initial specification, S_0, consists of the free constructors of *sequence*. Since there are no axioms in S_0, completeness of definition, consistency, termination, and sufficient completeness are all trivially satisfied. Now we extend S_0 by defining the operation *insert*. We define its left sides with the BCS. The right side of each axiom is obtained by functional composition of symbols in S_0 and possibly *insert*, the operation we are defining. The latter can occur only within the recursive reduction of the left side, when some argument is a recursive constructor. This yields an extension of S_0 that we denote with S_1. The good behavior of S_0 and the use of the strategies ensure the good behavior of S_1. Now we can similarly extend S_1 with the operation *sort* and obtain a new well-behaved specification.

Our design approach is not appropriate for every specification. When it can be used, however, the approach yields a specification with desirable properties that are in general undecidable and are not easy to verify in practice. If the signature contains non-free constructors, then to ensure the good behavior of the specification we must still verify its termination and consistency. For the latter it suffices to verify that every critical pair is convergent.

4 Direct implementation

4.1 Translation scheme

Rewriting is a model of computation. Specifications in the form of a rewrite system are executable — one simply rewrites terms to their normal forms. This implementation of a specification, known as *direct* [17], is relatively straightforward for convergent, constructor-based systems. The interest in directly implementing a specification stems from the possibility of executing the specified software without incurring the cost of developing the code. That is, the direct implementation of the specification is a software prototype.

Many specification environments supporting rewriting, e.g., [14, 16], offer this form of prototyping. A common limitation of these prototypes is that they can be executed only in the specification environment. If the prototype is activated by existing code, or uses object libraries, or interacts with the operating system, then it may become necessary to execute the prototype in the same environment that will host the final code. A solution to this problem consists in mapping the rewrite system to various computational paradigms or particular languages [2]. For example, in functional and procedural languages constructors are mapped into code that builds instances of dynamic data structures, whereas operations are mapped into subprograms. The description of the transformation of a specification in Prolog [11] is discussed next. We choose this language because it is well-suited for coding harnesses of software prototypes and for creating complex, structured data that exercise these prototypes. Prolog is also well-suited for symbolic manipulation. Many ideas discussed in this note, including the automated prover described in the next section, have been implemented in this language [3]. Adding the direct implementation to these tools makes a rich environment for reasoning and experimenting with specifications.

If f is an operation with n arguments, the direct implementation in Prolog of f is a predicate `f` with $n+1$ arguments. The additional argument of `f` is used for “returning” the result of f applied to the other arguments. Each axiom defining an operation yields a Horn clause. In order to describe the details of the translation, we introduce a few notational conventions. We overload the comma symbol to denote both separation of string elements and concatenation of strings, i.e., if $x = x_1, \ldots, x_i$ and $y = y_1, \ldots, y_j$ are strings, with $i, j \geq 0$, then $x, y = x_1, \ldots, x_i, y_1, \ldots, y_j$. If τ

is a function whose range is a set of non-null strings, then $\dot{\tau}(x)$ is the last element of $\tau(x)$, and $\bar{\tau}(x)$ is $\tau(x)$ without its last element. Combining the previous two notations, we have $\tau(x) = \bar{\tau}(x), \dot{\tau}(x)$.

The scheme for the direct implementation of a specification into Prolog is based on a function, τ, that maps terms of the specification into strings of Prolog terms. Symbols of the specification signature are mapped into Prolog symbols with the same spelling. The context of a symbol and the font in which is written, italic for the specification and typewriter for Prolog, resolve the potential ambiguity. `T` is a fresh Prolog variable.

$$\tau(t) = \begin{cases} \mathtt{X} & \text{if } t = X \text{ and } X \text{ is a variable;} \\ \bar{\tau}(t_1), \ldots, \bar{\tau}(t_k),\ \mathtt{c}(\dot{\tau}(t_1), \ldots, \dot{\tau}(t_k)) & \text{if } t = c(t_1, \ldots, t_k) \text{ and } c \text{ is a constructor;} \\ \bar{\tau}(t_1), \ldots, \bar{\tau}(t_k),\ \mathtt{f}(\dot{\tau}(t_1), \ldots, \dot{\tau}(t_k), \mathtt{T}),\ \mathtt{T} & \text{if } t = f(t_1, \ldots, t_k) \text{ and } f \text{ is an operation.} \end{cases}$$

Thus τ associates a Prolog predicate `f` to each operation f of the signature and an unevaluable symbol to each constructor. When it is extended from terms to axioms, τ yields Horn clauses.

$$\tau(f(t_1, \ldots, t_k) \to t) = \begin{cases} \mathtt{f}(t_1, \ldots, t_k, \dot{\tau}(t)). & \text{if } \bar{\tau}(t) \text{ is null;} \\ \mathtt{f}(t_1, \ldots, t_k, \dot{\tau}(t)) :- \bar{\tau}(t). & \text{otherwise.} \end{cases}$$

For example, to translate $\tau(\mathit{sort}(\mathit{cons}(E, L)) \to \mathit{insert}(E, \mathit{sort}(L)))$ to a Horn clause, we compute the following terms:

$$\begin{aligned} \tau(\mathit{insert}(E, \mathit{sort}(L))) &= \bar{\tau}(E), \bar{\tau}(\mathit{sort}(L)), \mathtt{insert}(\dot{\tau}(E), \dot{\tau}(\mathit{sort}(L)), \mathtt{T}), \mathtt{T} \\ &= \epsilon, \mathtt{sort(L,U)}, \mathtt{insert}(\dot{\tau}(E), \dot{\tau}(\mathit{sort}(L)), \mathtt{T}), \mathtt{T} \\ &= \mathtt{sort(L,U)}, \mathtt{insert}(\dot{\tau}(E), \dot{\tau}(\mathit{sort}(L)), \mathtt{T}), \mathtt{T} \\ &= \mathtt{sort(L,U)}, \mathtt{insert(E,U,T)}, \mathtt{T} \\ \tau(\mathit{sort}(L)) &= \bar{\tau}(L), \mathtt{sort}(\dot{\tau}(L), \mathtt{U}), \mathtt{U} \\ &= \epsilon, \mathtt{sort(L,U)}, \mathtt{U} \\ &= \mathtt{sort(L,U)}, \mathtt{U} \end{aligned}$$

Thus $\tau(\mathit{sort}(\mathit{cons}(E, L)) \to \mathit{insert}(E, \mathit{sort}(L)))$ yields:

```
sort(cons(E,L),T) :- sort(L,U), insert(E,U,T).
```

An actual translator handles Prolog predefined types, such as numbers, and the "**if-then-else**" operation in an ad-hoc manner. The cut improves the efficiency of the directly implemented code. It avoids checking $E > F$ in the third clause of `insert` below. The direct implementation of our first specification is:

```
sort(nil,nil).
sort(cons(E,L),T) :- sort(L,U), insert(E,U,T).
insert(E,nil,cons(E,nil)).
insert(E,cons(F,L),cons(E,cons(F,L))) :- E<=F, !.
insert(E,cons(F,L),cons(F,T)) :- insert(E,L,T), .
```

This implementation scheme is equivalent to [25], but the mapping τ provides more than a terse description of the transformation of a specification. An implementation of τ is retained in the prototype to provide a harness to invoke the directly implemented defined operations in a natural way. We define a predicate, `isab`, which plays, for abstract data types, the role played by the predefined predicate `is` for numeric types. `isab`, declared to be an infix operator with the same precedence and associativity of `is`, is (abstractly) implemented, using τ, as follows:

```
isab(τ̇(X),X) :- call(τ̄(X)).
```

For example, a harness for experimenting with the specifications discussed in the introduction may attempt to verify that the result of sorting a sequence is sorted. The harness creates some sequence s and executes:

```
N isab sorted(sort(s))
```

Expressions of this kind are simpler and more natural than the corresponding flattened expressions of [25] and decrease the possibility of introducing errors in the harness.

4.2 Using prototypes

We use the prototypes obtained by the direct implementation in two ways. The previous section shows the first possibility. The prototype allows us to verify the agreement between our intuitive expectations and the behavior or properties described by the specification. For example, we verify whether *sort* applied to a sequence indeed sorts it, or whether *sorted* returns *true* if and only if its argument is a sorted sequence. We can also combine the two specifications, as shown in a previous example, to check their mutual agreement. Our specification is too simple to be of practical interest, but in principle there are no scalability problems.

When a prototype interacts with code not generated by the direct implementation of a specification, it is generated in the language of this code. Well-behaved specifications, such as those produced using the strategies discussed earlier, are easy to translate into imperative and functional languages too [2].

The second possibility offered by the direct implementation of a specification originates from the use of the specification to check the correct execution of the code it specifies. For example, a C++ implementation of a class *sequence* has a method *sort* with the obvious meaning. The sortedness of a sequence object after sending it a *sort* message is a necessary condition for the correctness of the method. An assertion placed at the end of the method checks this condition:

```
void sequence::sort () {
  // body of the function
  assert (sorted (this object))
};
```

This use of specifications, discussed in [23] for the Ada language and in [24] for Eiffel, is effective, but has some drawbacks. Assertions involve methods of the class that may have to be coded just for this purpose, e.g., *sorted* or *permutation.* The same representation of *sequence* is used both by the operation being checked, *sort*, and by the operations checking it, *sorted.* Thus, loss of information in the representation may go undetected — for example the trivial type (which has a single value) satisfies any specification. Most important, it is not always easy to find properties that uniquely and completely characterize the behavior of an operation. What is missing in these approaches is a satisfactory degree of independence between the code and its assertions.

A more sophisticated approach [5] based on the direct implementation of a specification achieves this goal of independence. The specification is directly implemented with the intent of running it together with the code that it specifies. The direct implementation is no longer a prototype in the classical sense, but it coexists with the real code in the same run-time environment. The code uses the specification to check it itself. A significant advantage is using the specification of the behavior of a method, rather than the properties of its output. With some imprecision that we will correct shortly, the assertion for the *sequence* method *sort* becomes

`assert` (*this object* `==` `spec_sort` (*this initial object*))

where *spec_sort* is the name given to the direct implementation of the specification operation *sort* and "*this initial object*" is the state of the object before the execution of the *sort* method. Assertions of this kind are more convenient than those in [23, 24] and do not require coding operations that appear only in the assertions. However, we must retain the state of the object at the time the method *sort* is invoked, and we must deal with two different representations for the type *sequence*: the representation chosen by the class implementer, that we refer to as concrete, and the representation generated by the direct implementation scheme, that we refer to as abstract.

The solution discussed in [5] maintains both representations of an object. The test for equality performed by an assertion is between abstract representations. The concrete representation of an object at the end of the execution of a method is mapped to its abstract counterpart using what in [18] is called a representation mapping. If we denote this mapping with $\mathcal{A}$, the required assertion for the example we are discussing is:

`assert` ($\mathcal{A}$(*this concrete object*) `==` `spec_sort` (*this initial abstract object*))

The representation mapping $\mathcal{A}$ must be coded by the programmer. For example, suppose that the programmer represents a sequence using a dynamically allocated array. The instance variables of the class are the address of the array, the size of the array, and the number of elements in the sequence.

```
class sequence {
  private:
    int cursize;     // length of this sequence
    int maxsize;     // size of allocated array
    int * seq;       // address of array of elements
  //  Other private and public members
};
```

The representation mapping is shown below. The string "`spec_`" is prefixed to the symbols and the types directly implemented from the specification as opposed to those implemented by the programmer.

```
spec_sequence map (const sequence & s) {
  spec_sequence r = spec_nil ();
  for (int i = 0; i < s.cursize; i++) r = spec_cons (x[i], r);
  return r;
};
```

It is argued in [5] that the extra programming effort required to code a representation mapping is a blessing in disguise. Unless the programmer has an accurate idea of this mapping, he cannot write correct methods that implement the specification's operations. There is no better way to ensure this understanding than to insist that it be put into code and executed.

The equality in the assertion is between abstract objects and thus corresponds to the syntactic equality of normal forms. The concrete representation of an object is maintained by the program. The abstract representation is computed within the assertion, thus only a minimum of bookkeeping is necessary to maintain it for use at a later time.

The second technique discussed in this section contains aspects of both multi-version programming, when the results of the directly implemented specification and its implementation by a programmer are compared for concordance, and self-checking, when the assertions check the internal state of an object. Both techniques have been investigated [6, 20, 22] in relation to safety-critical applications.

5 Theorem proving

The most challenging application of a specification is to prove the correctness of software. We have experimented with an automated prover incorporating many concepts from the Boyer-Moore Theorem Prover [8]. However, except for built-in knowledge of term equality and data type induction, the knowledge in the theorem prover is supplied by specifications. The prover takes advantage of the characteristics of a specification designed with the strategies that we have discussed earlier.

The theorem prover computes a boolean recursive function, called *prove*, whose input is an equation and whose output is *true* if and only if the equation has been proved. Axioms and lemmas of the specification are accessed as global data. Proofs of theorems are generated as side effects of computations of *prove*. By default, the prover autonomously performs reductions, selects inductive variables, generates the induction cases, applies the inductive hypotheses, and generalizes formulas to be proved. Users may override the automatic choices, made by the prover, for inductive variables and generalizations. A technique discussed in [4] allows users to employ a limited form of case analyses in proofs.

The theorem prover executes four basic actions for proving an equation: *reduce*, *fertilize*, *generalize*, and *induct*. *Reduce* applies a rewrite rule to the formula being proved. *Fertilize* is responsible for "using" an inductive hypothesis, i.e., replacing a subterm in the current formula with an equivalent term from an inductive hypothesis. *Generalize* tries to replace some non-variable subterm common to both sides of the formula with a fresh variable. *Induct* selects an inductive variable and generates new equations.

```
function $prove(E)$ is
    begin
        if $E$ has the form $x = x$, for some term $x$, then return $true$; end if;
        if $E$ can be reduced, then return $prove(reduce(E))$; end if;
        if $E$ can be fertilized, then return $prove(fertilize(E))$; end if;
        $E' := generalize(E)$;
        if $E'$ contains an inductive variable, then
            $E_1, \ldots, E_n := induct(E')$;
            return $prove(E_1)$ andalso $\ldots$ andalso $prove(E_n)$;
        end if;
        return $false$;
    end $prove$;
```

An attempt to prove a theorem may exhaust the available resources, since induction may generate an infinite sequence of formulas to be proved. However, the termination property of the rewrite system guarantees that an equation cannot be reduced forever and the elimination of previously used inductive hypotheses [8] guarantees that an equation cannot be fertilized forever.

The strategies we have discussed ease some basic operations of the prover. The essential operations of every non-trivial proof involving an infinite type are *reduce* and *fertilize.* The BCS facilitates the first and the RRS the second.

The prover generates the formulas that constitute the cases of a proof by induction by instantiating an inductive variable with a constructor enumeration. When the inductive variable of a formula is an argument of an operation corresponding to an "*inductive*" choice in the operation's design with the BCS, the formulas that constitute the cases of a proof can all be immediately reduced.

The prover generates inductive steps of the form $l = r \supset l' = r'$. Fertilization is the process of replacing l with r in l' or, symmetrically, r with l in r'. If the inductive variable is selected according to the previous paragraph, then either l' or r' is reducible. If the axiom reducing one of these terms is designed with the RRS, then the reduced term can be fertilized.

5.1 An example

A common programming problem is the extraction of some information from a collection of values, for example, finding the smallest element. Let C be the collection and T the type of its elements. An informal specification of the above problem contains three steps. First, put aside one element of C, say f; second, find the smallest element, say g, of C without f; and third, return the smallest among f and g. This specification is just an instance of a computational paradigm known as *accumulation* [7]. An abstract, general accumulator is equationally specified by the function A defined next. The sequence $e_1, e_2, \ldots$ is a presentation in some order of the collection to be accumulated.

$$A(e_i, \ldots, e_j) = \begin{cases} init, & \text{if } i > j; \\ step(A(e_i, \ldots e_{j-1}), e_j), & \text{otherwise.} \end{cases}$$

We can instantiate our example by defining the type *collection* with two constructors, *empty,* with the obvious meaning, and *add,* which takes an element e and a collection c and constructs the collection obtained by adding e to c. Replacing *step* with *min*, a function computing the minimum of two arguments, and *init* with *maxnat*, a special token discussed shortly, results in the following equations.

$$\begin{aligned}&\mathit{accum}(\mathit{empty}) = \mathit{maxnat}\\&\mathit{accum}(\mathit{add}(E, C)) = \mathit{min}(E, \mathit{accum}(C))\end{aligned}$$

The symbol *maxnat* stands for the exception raised by the computation of the minimum of an empty collection. Order sorted specifications [15] give a precise meaning to this construction. The operation *min* is defined on a supersort of the natural numbers that contains *maxnat*, too. When one argument of *min* is *maxnat*, the operation returns the other argument. Thus, *min* sees *maxnat* as the "maximum natural number" and handles the exception. *Min* also propagates an exception if the other argument is *maxnat*, too.

Variants of this scheme can be used for a variety of other tasks, e.g., to add or multiply together the elements of the collection, when they are numbers, or to count how many elements satisfy a certain property, etc. The instantiations of *init* and *step* for a specific problem are generally easy to find, but require some care.

Our specification starts the accumulation from the initial portion of the presentation, as would be appropriate for a collection represented by an array. A different representation of the collection, e.g., a linked list, would suggest to start the accumulation from the other end. The second case of the definition of A would then be:

$$\mathit{step}(e_i, A(e_{i+1}, \ldots e_j))$$

While for our example the two versions are interchangeable; for other problems, such as converting a string of digits in the integer represented by the string, one version may be incorrect or inconvenient.

In general, if we want to prove the correctness of a piece of software or increase our confidence in the appropriateness of using an accumulator for a certain computation, we may have to reason about these definitions. For example, we may wish to prove that the two accumulations are equivalent or to verify whether the order in which the elements of the collection are accumulated affects the result of the accumulation.

The prover helps us accomplish these tasks. For example, we may prove that the equivalence of the above accumulation is implied by the following more general property:

$$A(e_i, \ldots, e_j) = \mathit{step}(A(e_i, \ldots, e_k), A(e_{k+1}, \ldots, e_j)) \qquad i \le k \le j$$

A condition sufficient to ensure the validity of this equation is that $(T; \mathit{step}, \mathit{init})$ is a left monoid. Thus, we may attempt to prove the above equations for some problem-specific instantiations of *init* and *step*.

In the Appendix we discuss the specification of this problem in a form suitable for our prover and show the formal proofs that we obtain. The analysis of this problem suggests that may be possible to implement some accumulations in parallel or by a divide-and-conquer technique. A further discussion on this opportunity appears in [4].

6 Concluding remarks

We discussed some design techniques for an equational specification and some uses of the specification for software development. The specifications are in the form of constructor-based rewrite systems. The design techniques consists of two strategies and an incremental extension approach that allow us to produces specifications with desirable properties that are generally undecidable and difficult to verify in practice. The properties provided by our techniques are completeness of definitions, consistency, termination, and sufficient completeness. They also imply confluence and uniqueness of normal forms. The strategies are simple in practice and can be automated to a considerable degree. These strategies cannot be used for any specification, but when they can be used, they help in producing high quality specifications with relative ease.

Specifications with the properties provided by the strategies can be directly implemented in a variety of computing paradigms representative of mainstream programming languages. The direct implementation of a specification is modeled by rewriting. The resulting executable code can be activated by a harness that also provides test data and/or allows the inspection of the result of a computation. In this way we test whether our intuitive understanding of a problem is accurately captured by a formal specification.

A more sophisticated application allows a program to use a directly implemented specification as an oracle for self-checking. Self-checking is useful during the testing and debugging phases of the software lifecycle, since it minimizes oversights of the testing and debugging teams, it checks the internal states of objects in addition to their input/output relations, and it accurately localizes the region of a program where an error has occurred.

Specifications with the properties provided by the strategies facilitate reasoning about the specification themselves and about the correctness of programs. The BCS makes it easy to find inductive variables and to reduce all the cases of an induction. This, in conjunction with the RRS, creates in a formula the conditions for applying an inductive hypothesis. We cannot quantify the impact of these facts on proving theorems, although they appear to be quite useful.

Automated theorem proving is a very hard task. Our approach aims more at managing the complexity of a problem than on relying on the power of a prover. The theorem prover is conceptually simple. The printouts produced by the prover are easy to understand, and the user can follow the steps of a proof and the reasons of the prover for executing them. This is essential for discovering lemmas that help completing a difficult proof or for discovering that a specification does not capture the intuitive understand of a problem. Although we can prove only very simple theorems, the prover expands our ability to reason about specifications and programs, and complements the opportunities provided by the direct implementation.

Acknowledgment

We would like to thank A. Miola for coordinating the TASSO project and all the members of the project for stimulating discussions.

References

[1] S. Antoy. Design strategies for rewrite rules. In S. Kaplan and M. Okada, editors, *CTRS'90*, pages 333–341, Montreal, Canada, June 1990. Lect. Notes in Comp. Sci., Vol. 516.

[2] S. Antoy, P. Forcheri, and M.T. Molfino. Specification-based code generation. In *23rd Hawaii Int'l Conf. on System Sciences*, pages 165–173, Kailua-Kona, Hawaii, Jan. 3-5 1990.

[3] S. Antoy, P. Forcheri, M.T. Molfino, and C. Schenone. A uniform approach to deduction and automatic implementation. In *DISCO'92*, pages 29–48, Bath, UK, April 13-15 1992. Lect. Notes in Comp. Sci., Vol. 721.

[4] S. Antoy and J. Gannon. Using term rewriting systsem to verify software. *IEEE Trans. Soft. Eng.*, 20(4):259–274, 1994.

[5] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specifications. In *2nd Irvine Software Symposium*, pages 29–48, Irvine, CA, March 6 1992.

[6] A. Avizienis and J. Kelly. Fault tolerance by design diversity: concepts and experiments. *Computer*, 17:67–80, 1984.

[7] S.K. Basu. On development of iterative programs from functional specifications. *IEEE Trans. Soft. Eng.*, 6(2):170–182, 1980.

[8] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[9] R. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[10] C. Choppy, S. Kaplan, and M. Soria. Complexity analysis of term-rewriting systems. *Theoretical Computer Science*, 67:261–282, 1989.

[11] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, second edition, 1984.

[12] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[13] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. Springer-Verlag, Berlin, 1985.

[14] S.J. Garland, J.V. Guttag, and J.J. Horning. Debugging Larch shared language specifications. *IEEE Trans. on Soft. Eng.*, 16(9):1044–1057, 1990.

[15] J.A. Goguen. Order sorted algebras. Technical Report 14, Computer Science Department, UCLA, 1978.

[16] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, CA, 1988.

[17] J.V. Guttag, E. Horowitz, and D. Musser. Abstract data types and software validation. *Comm. of the ACM*, 21:1048–1064, 1978.

[18] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[19] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 1–112. Oxford University Press, 1992.

[20] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on Soft. Eng.*, 12:96–109, 1986.

[21] D.E. Knuth and P.B. Bendix. *Simple word problems in universal algebras*, pages 263–297. Pergamon, 1970.

[22] N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The use of self checks and voting in software detection: An empirical study. *IEEE Trans. on Soft. Eng.*, 16:432–443, 1990.

[23] D.C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs.* Springer-Verlag, Berlin, 1990.

[24] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, 1988.

[25] M.H. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 4:265–288, 1987.

Appendix

We formalize the accumulator example discussed in the text as follows. The type *collection* has two constructors, *empty*, with the obvious meaning, and *add*, which takes an element e and a collection c and constructs the collection obtained by adding e to c.

Collection can be *concat*enated and *accum*ulated, where the definition of each operation is given by the axioms below. We label the axioms to reference their applications in a proof printout.

$concat(empty, C) = C$	concat_1
$concat(add(E, C), D) = add(E, concat(C, D))$	concat_2
$accum(empty) = init$	accum_1
$accum(add(E, C)) = step(E, accum(C))$	accum_2

The elements of a collection and the symbols *init* and *step* are generic, i.e., they are instantiated only for a specific problem, no constructors are defined for the type *element* and there are no defining axioms for the operations. We want to prove that if the type *element* with this operation is a left monoid, then the order in which the elements of a collection are accumulated does not affect the result of the accumulation. Hence, we assume:

$step(init, X) = X$	left identity
$step(step(X, Y), Z) = step(X, step(Y, Z))$	associativity

and we attempt to prove:

$$accum(concat(X, Y)) = step(accum(X), accum(Y))$$

The prover prints the following proof, where `A0`, `A1`, ... are the names of variables generated by the prover. Lines headed by "`IH-`" show inductive hypotheses, if any. Lines headed by "`(L)`" or "`(R)`" indicate the application of a transformation to the left or respectively right side of the equation being proved. The transformation is explained in the string delimited by "`<<`."

```
The theorem is:
  accum(concat(A0,A1)) = step(accum(A0),accum(A1))
Begin induction on A0
Induction on A0 case empty
Inductive hypotheses are:
  (L) accum(concat(empty,A1))     << subst A0 with empty <<
  (R) step(accum(empty),accum(A1))     << subst A0 with empty <<
  (L) accum(A1)     << reduct by concat_1 <<
  (R) step(init,accum(A1))     << reduct by accum_1 <<
  (R) accum(A1)     << reduct by left identity <<
  *** equality obtained ***
Induction on A0 case add(A2,A3)
Inductive hypotheses are:
  IH- accum(concat(A3,A1))=step(accum(A3),accum(A1))
  (L) accum(concat(add(A2,A3),A1))     << subst A0 with add(A2,A3) <<
  (R) step(accum(add(A2,A3)),accum(A1))     << subst A0 with add(A2,A3) <<
  (L) accum(add(A2,concat(A3,A1)))     << reduct by concat_2 <<
  (L) step(A2,accum(concat(A3,A1)))     << reduct by accum_2 <<
  (R) step(step(A2,accum(A3)),accum(A1))     << reduct by accum_2 <<
  (R) step(A2,step(accum(A3),accum(A1)))     << reduct by associativity <<
  (L) step(A2,step(accum(A3),accum(A1)))     << ind. hyp. on  A0 for A3 <<
  *** equality obtained ***
End induction on A0
QED
```

If we have an instance of an accumulation and, for example, we want to implement it in parallel, it suffices to verify the left monoid property. Suppose that the problem is to find the minimum element of a collection. The type of the collection's elements is a subtype (supersort) of the natural numbers. The natural numbers are constructed by 0 and *succ* as usual. The supersort contains one extra element, *maxnat*. The axioms defining the operation *min* on this type are:

$min(0, X) = 0$	min 1
$min(succ(X), 0) = 0$	min 2
$min(succ(X), succ(Y)) = succ(\$)$	min 3
$min(succ(X), maxnat) = succ(X)$	min 4
$min(maxnat, X) = X$	min 5

Thus, we have to prove that *maxnat* is a left identity of *min* and that *min* is associative, i.e.:

$$min(maxnat, X) = X$$
$$min(min(X, Y), Z) = min(X, min(Y, Z))$$

The theorem prover prints, without assistance, the following proofs. The first is trivial. The second proof shows a triple nested induction.

```
The theorem is:
  min(maxnat,A0) = A0
  (L) A0     << reduct by min 5 <<
  *** equality obtained ***
QED
```

```
The theorem is:
  min(min(A1,A2),A3) = min(A1,min(A2,A3))
Begin induction on A1
Induction on A1 case 0
Inductive hypotheses are:
  (L) min(min(0,A2),A3)     << subst A1 with 0 <<
  (R) min(0,min(A2,A3))     << subst A1 with 0 <<
  (L) min(0,A3)     << reduct by min 1 <<
  (L) 0     << reduct by min 1 <<
  (R) 0     << reduct by min 1 <<
  *** equality obtained ***
Induction on A1 case succ(A4)
Inductive hypotheses are:
  IH- min(min(A4,A2),A3)=min(A4,min(A2,A3))
  (L) min(min(succ(A4),A2),A3)     << subst A1 with succ(A4) <<
  (R) min(succ(A4),min(A2,A3))     << subst A1 with succ(A4) <<
Begin induction on A2
Induction on A2 case 0
Inductive hypotheses are:
  IH- min(min(A4,A2),A3)=min(A4,min(A2,A3))
  (L) min(min(succ(A4),0),A3)     << subst A2 with 0 <<
  (R) min(succ(A4),min(0,A3))     << subst A2 with 0 <<
  (L) min(0,A3)     << reduct by min 2 <<
  (L) 0     << reduct by min 1 <<
  (R) min(succ(A4),0)     << reduct by min 1 <<
  (R) 0     << reduct by min 2 <<
  *** equality obtained ***
Induction on A2 case succ(A5)
Inductive hypotheses are:
  IH- min(min(A4,A5),A3)=min(A4,min(A5,A3))
  IH- min(min(succ(A4),A5),A3)=min(succ(A4),min(A5,A3))
  (L) min(min(succ(A4),succ(A5)),A3)     << subst A2 with succ(A5) <<
  (R) min(succ(A4),min(succ(A5),A3))     << subst A2 with succ(A5) <<
  (L) min(succ(min(A4,A5)),A3)     << reduct by min 3 <<
Begin induction on A3
Induction on A3 case 0
Inductive hypotheses are:
  IH- min(min(A4,A5),A3)=min(A4,min(A5,A3))
  IH- min(min(succ(A4),A5),A3)=min(succ(A4),min(A5,A3))
  (L) min(succ(min(A4,A5)),0)     << subst A3 with 0 <<
  (R) min(succ(A4),min(succ(A5),0))     << subst A3 with 0 <<
  (L) 0     << reduct by min 2 <<
  (R) min(succ(A4),0)     << reduct by min 2 <<
  (R) 0     << reduct by min 2 <<
  *** equality obtained ***
Induction on A3 case succ(A6)
Inductive hypotheses are:
  IH- min(min(A4,A5),A6)=min(A4,min(A5,A6))
  IH- min(min(succ(A4),A5),A6)=min(succ(A4),min(A5,A6))
  IH- min(succ(min(A4,A5)),A6)=min(succ(A4),min(succ(A5),A6))
  (L) min(succ(min(A4,A5)),succ(A6))     << subst A3 with succ(A6) <<
  (R) min(succ(A4),min(succ(A5),succ(A6)))     << subst A3 with succ(A6) <<
  (L) succ(min(min(A4,A5),A6))     << reduct by min 3 <<
  (R) min(succ(A4),succ(min(A5,A6)))     << reduct by min 3 <<
  (R) succ(min(A4,min(A5,A6)))     << reduct by min 3 <<
  (L) succ(min(A4,min(A5,A6)))     << ind. hyp. on  A1 for A4 <<
  *** equality obtained ***
Induction on A3 case maxnat
Inductive hypotheses are:
  IH- min(min(A4,A5),A3)=min(A4,min(A5,A3))
  IH- min(min(succ(A4),A5),A3)=min(succ(A4),min(A5,A3))
  (L) min(succ(min(A4,A5)),maxnat)     << subst A3 with maxnat <<
  (R) min(succ(A4),min(succ(A5),maxnat))     << subst A3 with maxnat <<
  (L) succ(min(A4,A5))     << reduct by min 4 <<
  (R) min(succ(A4),succ(A5))     << reduct by min 4 <<
  (R) succ(min(A4,A5))     << reduct by min 3 <<
  *** equality obtained ***
End induction on A3
Induction on A2 case maxnat
Inductive hypotheses are:
  IH- min(min(A4,A2),A3)=min(A4,min(A2,A3))
  (L) min(min(succ(A4),maxnat),A3)     << subst A2 with maxnat <<
  (R) min(succ(A4),min(maxnat,A3))     << subst A2 with maxnat <<
  (L) min(succ(A4),A3)     << reduct by min 4 <<
  (R) min(succ(A4),A3)     << reduct by min 5 <<
  *** equality obtained ***
End induction on A2
Induction on A1 case maxnat
Inductive hypotheses are:
  (L) min(min(maxnat,A2),A3)     << subst A1 with maxnat <<
  (R) min(maxnat,min(A2,A3))     << subst A1 with maxnat <<
  (L) min(A2,A3)     << reduct by min 5 <<
  (R) min(A2,A3)     << reduct by min 5 <<
  *** equality obtained ***
End induction on A1
QED
```