Automatically Checking an Implementation against Its Formal Specification

Sergio Antoy[*] Dick Hamlet[†]

Portland State University
Department of Computer Science
and
Center for Software Quality Research
Portland, OR 97207

January 27, 1999

Abstract

We propose to check the execution of an abstract data type's imperative implementation against its algebraic specification. An explicit mapping from implementation states to abstract values is added to the imperative code. The form of specification allows mechanical checking of desirable properties such as consistency and completeness, particularly when operations are added incrementally to the data type. During unit testing, the specification serves as a test oracle. Any variance between computed and specified values is automatically detected. When the module is made part of some application, the checking can be removed, or may remain in place for further validating the implementation. The specification, executed by rewriting, can be thought of as itself an implementation with maximum design diversity, and the validation as a form of multiversion-programming comparison.

Index Terms

Self-checking Code, Object-Oriented Software Testing, Formal Specification, Rewriting.

[*] Supported by National Science Foundation grant CCR-9406751.

[†] Supported by National Science Foundation grant CCR-9110111.

1 Introduction

Encapsulated data abstractions, also called abstract data types (ADTs), are the most promising programming-language idea to support software engineering. The ADT is the basis for the "information-hiding" design philosophy [52] that makes software easier to analyze and understand, and that supports maintenance and reuse. There are several formal specification techniques for ADTs [31], a growing number of language implementations of the idea [28], and accepted theories of ADT correctness [21, 27, 29, 58]. The ADT is a good setting for work on unit testing and testability [36].

However, for all the ADT's promise, fundamental problems remain concerning ADTs, their specifications, and implementations. In this paper we address the problem of checking agreement between an ADT's formal specification and its implementation. We use a particular kind of equational specification (as rewrite rules) [4], and an imperative implementation language. For examples of the latter we choose C++ [56] and Java [8] but almost any ADT programming language, e.g., Ada or Eiffel, would work as well. We do use special properties of one kind of rewrite specification, which would make it more difficult to substitute a different kind of specification part. We show how to write C++ or Java classes and their formal specifications so that the implementation is automatically checked against the specification during execution. Thus the specification serves as an "effective oracle," and in any application the self-checking ADT cannot deviate from specified behavior without the failure being detected.

Because the specification oracle may be inefficient in comparison to the imperative implementation, its use may be confined to prototyping and the testing phase of software development. However, for some applications the specification and implementation may be viewed as independent "versions" of the same software, which continually check each other. Since both were produced by human beings, both are subject to error, but their utterly different form and content suggest that there is minimum chance of a common-mode failure [45].

The central idea that allows self checking is to implement, as part of the imperative code, the mapping between concrete implementation states and abstract specification objects. Failure to mechanically capture this important part of design is a weakness of all existing ADT design systems.

2 Self-checking ADTs

We describe ADTs that check their implementations against specifications at run time, and give, in this section, a simple illustrative example in C++. In an appendix we present a richer example implemented in Java.

2.1 Automatic Testing

The implementation side of our approach is available "off the shelf," C++ in the running example. Any programming language that supports ADTs can be used—we do not need the "inheritance" part of an object-oriented language. One component of our scheme is thus a C++ class implementation. (Whether the implementation is thought of as arising from an intuitive set of requirements, or from a formal specification that is the second component of our scheme, is immaterial to this description; however, we discuss the issue in section 5.2.)

The second component we require is a formal specification of the axiomatic variety. Here we do not have so much leeway, because the specification form determines our ability to mechanically check for specification properties like the consistency of a newly added operation, and plays an essential role in efficiently checking for abstract equality when the specification serves as an implementation oracle. We choose to use a rewrite system restricted so that the desirable properties of confluence and termination can be largely obtained from the syntactic forms [4]. It would be possible to employ more general specifications, but at the cost of using more powerful (and less efficient) theorem provers. The limited goal of our scheme argues against this generality and loss of efficiency. We have made the engineering decision to use a specification fitted to the role of automatic oracle.

The user of our approach must supply one additional component, of central importance in our scheme: the user must write explicit code for a "representation" mapping[1] between the concrete data structures of C++ instance variables, and the abstractions of the specification. It is a major weakness of present ADT theories and methodologies that the form of the representation mapping is not explicitly prescribed. The existing theory is framed so that the implementation is correct if *there exists* a representation mapping, as a mathematical object [23]. We believe that coding the representation mapping in the implementation language is the right way to formally capture this component of data abstraction. In practice, the implementor must have a representation in mind, since it is one of the earliest design decisions. Coding the representation is solid documentation, and is typically very easy compared to the remainder of the implementation.

Having written an axiomatic specification, a C++ class, and an explicit representation mapping, the user may now test the composite ADT using our scheme and any unit-test technique. For example, a conventional driver and code coverage tool could be used to ensure that the C++ code has been adequately tested according to (say) a dataflow criterion [54]. Or, tests could be generated to exercise traces of class operations [36]. Whatever techniques are used, our approach will serve as the automatic test oracle that all existing testing systems lack. It determines the correctness of each operation invoked, according to the specification. Alternately, the user might decide to test the ADT "in place," by writing application code using it, and conducting a system test (perhaps using randomly generated inputs according to an expected operational profile [49]) on the application program with embedded ADT. During this test, our scheme ensures that the ADT cannot fail without detection.

2.2 Example: A Small Integer Set

To illustrate our ideas, we use the class "small set of integer" (first used by Hoare [35] to discuss the theory of data abstraction). To limit the details required to present our ideas, we keep this example as small as feasible. The application of our ideas to a piece of industrial software is described in the appendix. The signature for this ADT is shown in figure 1 (from Stroustrup [56]).

The signature diagram captures the complete syntax of ADT usage. For example, in figure 1 `empty` takes two arguments, an `elem` and a `nat`, and the operations for these types are also shown in the diagram.

[1] This name was used by Hoare in his foundational paper [35]. Perhaps "abstraction mapping" is the more common name, which also better expresses the direction.

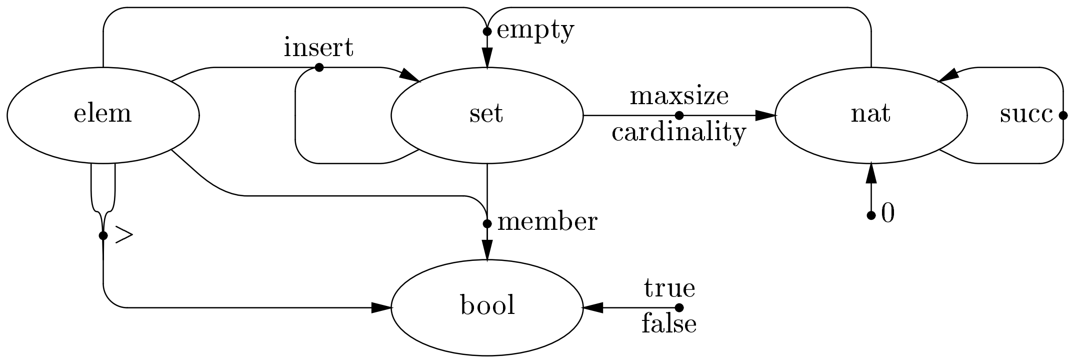

Figure 1: Signature of a generic, bounded ADT *set*.

2.2.1 The Specification

The semantics of an ADT can be specified with a set of equations (also called axioms) relating the ADT's operations. For example, the intuitive axioms for the insertion operation on a set are:

$$\begin{array}{l} insert(X, insert(Y, Z)) = insert(Y, insert(X, Z)) \\ insert(X, insert(X, Y)) = insert(X, Y). \end{array} \tag{1}$$

These equations determine which combinations of operations are equivalent, and along with an assumption about combinations not directly described by the equations, determine exactly which objects (expressed as ground terms using the operations) constitute the set which forms the ADT. The most common assumption is the *initial* one [27]: that any objects which cannot be proved equal using the equations are distinct. A good deal of work has been done on algebraic specification; see [13].

When equations are viewed as rewrite rules, the proofs of equivalences are simplified. In this view care must be taken that a rewrite system is a correct implementation of an algebraic specification. For this purpose, it suffices to consider ground-convergent systems, i.e., systems that are terminating and confluent for ground terms (see section 4 for a definition of these properties.) The above equations will not do as rewrite rules, because the first rule can be applied infinitely often. A suitable rewrite system can be obtained from an equational specification by completion [43], although the procedure is not guaranteed to terminate. Alternately, the form of the equations can be suitably restricted [4], which is the approach chosen here. Furthermore, most implementations of sets will impose bounds on both set size and the values of the set's elements, as the type *intset* given by Stroustrup [56, §5.3.2] does. The simple equations above do not describe these "practical" set ADTs.

We thus specify the ADT *intset*, from our understanding of the code in [56], using a notation similar to those of several other "algebraic" specifications, such as Act One [21], Asf [12] or Larch [30], and explain only those parts needed to understand the example. User-defined sorts are specified by an enumeration of constructors each with its arity and parameter types, followed by axioms. An axiom is a rewrite rule of the form

$$l \rightarrow r :- c, \tag{2}$$

where l and r are respectively the left and right sides of the rule and c is an optional condition [41]—a guard for the application of the rule. The symbol "?" on the right side of a rule denotes an exceptional result. Its semantics can be formalized within the framework of *order sorted algebras* [26]. For our more modest purposes, "?" denotes a computation that must be aborted and any term with an occurrence of "?" implicitly rewrites to "?". If the condition of a rule evaluates to "?" the rule is not fired. Following Prolog conventions [16] the identifier of a variable begins with an upper case letter and the underscore symbol denotes an anonymous variable.

The ADT *intset* of [56] is specified as follows:

```
sort intset
  constructor
    empty(integer,integer)         -- maxsize and element upper bound
    insert(integer,intset)         -- add element to set
  axiom
    empty(M,R) -> ? :- M<1 or R<M
    insert(E,empty(_,R)) -> ? :- E > R
    insert(E,insert(F,S)) -> insert(F,S) :- member(E,insert(F,S))
    insert(E,insert(F,S)) -> ?
      :- not member(E,insert(F,S))
         and cardinality(insert(F,S)) >= maxsize(insert(F,S))
    insert(E,insert(F,S)) -> insert(F,insert(E,S))
      :- not member(E,insert(F,S))
         and cardinality(insert(F,S)) < maxsize(insert(F,S))
         and E > F
```

The operations `member`, `cardinality`, and `maxsize` will be axiomatized shortly. We rely on the reader's intuition of these auxiliary concepts to explain the above axioms. The style used in these axioms handles exceptional cases with a "?" axiom, guarded by a constraint defining the exceptional condition. Thus in the `empty` axiom the exception occurs only if the parameters are inconsistent. Similarly, the first axiom for `insert` handles the error case in which an attempt is made to insert an element that violates the upper-bound restriction; and, the third axiom for `insert` handles an attempt to insert a new element into a set that is already at maximum size. The second and fourth `insert` axioms establish that the normal form for a nest of insertions is in element order, without duplicates. We could replace the second axiom of `insert` with

```
insert(E,S) -> S :- member(E,S)
```

and similarly the third axiom without changing the semantics of the specification. Although the axiom would be simpler, the overall specification would be more difficult to understand because the left hand sides of the first two axioms of `insert` would overlap.

Even in the given form, our axioms create some *overlaps*, i.e., the left hand side of some axiom unifies with a non-variable subterm of the left hand side of some other axiom. From any such overlap we obtain a *critical pair*; that is, the pair of terms created by rewriting the overlap with each of the two axioms. If the terms of a critical pair can both be rewritten to a common term, they are said to be *joinable*. If all critical pairs of a set of axioms are joinable, the axioms have the property of confluence. When the axioms are conditional, one only has to consider critical pairs for which the correspondingly instantiated conditions are not mutually exclusive; otherwise, they trivially satisfy the joinability test [19]. For example, the last three axioms of `insert` overlap, but their conditions are obviously mutually exclusive. By strengthening the conditions, more critical pairs become trivially joinable, but the specification becomes less intuitive and readable.

Specifications can grow incrementally by adopting the "stepwise specification by extension" approach [21]. Each increment adds new operations to a specification. The new specification is a *conservative extension* of the old one, i.e., it is complete and consistent. *Complete* means that the extension does not create new data elements. In other words, a term built with the new operations is equal (reducible) to some term of the original specification. *Consistent* means that the extension does not identify old data elements. In other words, terms that were not equal (reducible to a same term) in the original specification, do not become equal in the extension. We adopt two design strategies [4] to guarantee completeness and consistency, as follows:

- The *binary choice* strategy generates a set of complete and mutually exclusive arguments for an operation. Once we have a left side of a rule we define a set of right sides such that the set of conditions associated with the right sides are mutually exclusive.
- The *recursive reduction* strategy uses a mechanism similar to primitive recursion, but more expressive, for defining the right side of a rule in a way which ensures termination. The symbol "`!`" in the right side of a rule stands for the term obtained from the left side by replacing a term rooted by a recursive constructor with its recursive argument.

The only recursive constructor in the specification of *intset* is *insert*. The recursive argument of *insert* is the second one. Thus, the symbol "!" in the right-hand side of a rule with left-hand side of the form $f(\cdots insert(e,s)\cdots)$ stands for $f(\cdots s\cdots)$. The right-hand side is obtained from the left-hand side by replacing the subterm $insert(e,s)$ with s; every other portion of the left-hand side is unchanged. For example, "`!`" in the axioms of `cardinality` below stands for `cardinality(S)`.

The promised axioms for `cardinality`, `maxsize` and `member` are as follows:

```
operation cardinality(intset) -> integer
  axiom
    cardinality(empty(_,_)) -> 0
    cardinality(insert(E,S)) -> ! :- member(E,S)
    cardinality(insert(E,S)) -> succ(!) :- not member(E,S)

operation maxsize(intset) -> integer
  axiom
    maxsize(empty(M,_)) -> M
    maxsize(insert(_,_)) -> !

operation member(integer,intset) -> boolean
  axiom
    member(_,empty(_,_)) -> false
    member(E,insert(F,_)) -> E = F or !
```

(Recall that any term with an occurrence of "`?`" rewrites to "`?`;" hence, if the arguments to `empty` are indeed illegal, for example, each of these raises an exception.) This completes the example specification of `intset`.

2.2.2 Implementations of the Specification

We are going to discuss various implementations of the *set* specification of figure 1. The implementation language, C++, prompts us to implement a slightly different specification. This difference is irrelevant to the ideas that we are presenting and allows us to avoid unncessary complications that would hinder comprehension. C++ is (relatively speaking) a strongly typed language. Thus, we are forced to choose a type for the elements of a set. We choose the language's primitive type *int.* The implementation of a template class would be closer to the specification at the cost of additional complexity, but the additional generality would not be significant to our discussion. Also, in an implementation we replace the type `nat` of the specification with the language's primitive type *int* as well. The C++ programming language does not provide a natural number type and defining and using such a type in an implementation would again add complexity with very little gain.

We consider three implementations of this slightly specialized version of the *set* specification. They are referred to as a *by-hand implementation*, a *direct implementation*, and a *self-checking implementation.* A *by-hand implementation* is C++ code written by a programmer to provide the functionality expressed by the specification. This code is naturally structured as a C++ *class* in which operations are implemented as class member functions. A by-hand implementation of *intset* appears as the first example in Stroustrup's text [56, §5.3.2]. The *direct implementation* [31] is C++ code automatically generated from the specification by representing instances of abstract data types as terms, and manipulating those terms according to the rewrite rules. The *self-checking implementation* is the union of the by-hand implementation and the direct implementation with some additional C++ code to check their mutual agreement.

We describe the self-checking implementation first, even though it uses the direct implementation. This presentation order motivates the need for the direct implementation before its considerable detail is given.

The Self-checking Implementation

As described in the following section, the direct implementation provides the mechanism, in C++, for computing normal-form terms corresponding to any sequence of specification operations. The by-hand implementation provides a similar mechanism for computing a result using any sequence of its member-function invocations. These two computations correspond respectively to the upper (abstract) and lower (concrete) mappings in diagrams such as figure 2.

In the abstract world, $\in$ is the binary relation for set membership. In the concrete world, `member` is an operation that transforms the values of state variables. If starting at the lower left of the diagram and passing to the upper right by the two possible paths always yields the same result, we say that the diagram *commutes.* The concrete implementation in a commuting diagram is by definition correct according to the abstract specification there. Think of a state σ as a mapping from instance-variable names to their values. In any diagram with abstract operation F and concrete operation *f*, if the variables of the state are $x_1, x_2, ..., x_n$, the diagram commutes iff

$$\forall x_1 ... \forall x_n [F(R(\sigma(x_1)), ..., R(\sigma(x_n))) = R(f(\sigma(x_1), ..., \sigma(x_n)))]. \tag{3}$$

In figure 2, suppose that r is the value returned by `member` applied to `integer` variable x and `intset` variable S. Then the diagram commutes iff

$$R(\sigma(x)) \in R(\sigma(S)) \Leftrightarrow R(r) \tag{4}$$

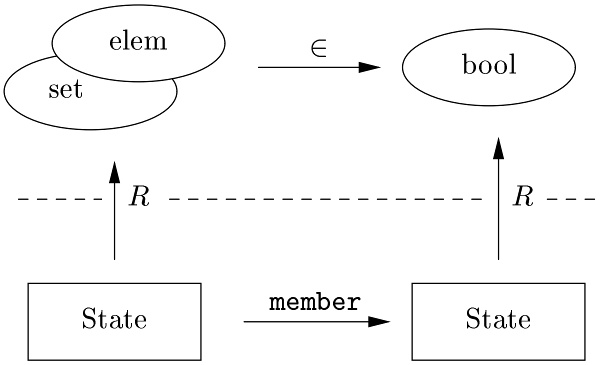

Figure 2: Commuting diagram for the `member` operation of the ADT *set*.

To check the by-hand result against the direct result only requires that code be available for the representation function to complete a diagram such as figure 2. The self-checking implementation comprises the C++ code of both by-hand and direct implementations, plus a representation function, and appropriate calls connecting them. The locus of control is placed in the by-hand implementation. When its member functions are invoked, corresponding normal-form terms are requested from the direct implementation. The comparison of results, however, takes place in the abstract world; what is actually compared are the normal forms computed there.

The self-checking implementation for *intset* illustrates this structure. A self-checking class has two additional private entities declared first, in the example of type `absset`, which is the type mark for a set in the direct implementation. The additional variable `abstract` contains values of sets from the direct implementation. The additional function `conc2abstr` is the representation mapping; it takes as input parameters the instance variables of `intset` and returns the corresponding `absset` instance.

```
// Declaration of the self-checking intset class.
class intset {
  absset abstract;                  // abstract version of this class
  absset concr2abstr();             // representation function
// Below this line the class is identical to Stroustrup, p. 146ff
  int cursize, maxsize;
  int *x;
public:
  intset(int m, int n);             // at most m ints in 1..n
  ...                               // (most of the code omitted)
```

Member functions of the self-checking implementation differ from the corresponding ones in the by-hand implementation only by the addition of two statements just before each function return. For example, the self-checking member function implementing the specification operation `empty` follows:

```
intset::intset(int m, int n)        // at most m ints in 1..n
```

```
{
  if (m<1 || n<m) error("illegal intset size");
  cursize = 0;
  maxsize = m;
  x = new int[maxsize];
// Additional statements for self-checking
  abstract = empty(m,n);              // compute abstract value
  verify;                             // check mutual agreement
}
```

The direct-implementation function empty is called and its result—a normal form encoded in the data structures of the direct implementation—saved in the added global variable abstract of type absset. (The code for the direct implementation of member function empty, also of type absset, appears in the following section.)

The macro verify performs an equality test between the value stored in abstract and that computed by conc2abstr, which is also a normal-form value. This equality test is particularly simple because in the direct implementation equality means identity of normal forms. The verify macro includes an error report if the two values differ. Its code follows.

```
#define verify \
  if (! equal(abstract,concr2abstr())) \
      cerr << form("Direct differs from by-hand at line %d of '%s'.\n", \
                   __LINE__, __FILE__)
```

The last significant piece of code of the self-checking implementation is the representation function. The mapping is straightforward, starting with an empty abstract set and adding elements from the concrete version one at a time to calculate the corresponding abstract set.

```
absset intset::concr2abstr()
{
  absset h = empty(maxsize,MAXINT);      // upper bound not implemented!
  for (int i = 0; i < cursize; i++) h = ::insert(x[i],h);
  return h;
}
```

It was only when writing this function that the programmer noticed that the by-hand implementation [56] pays no attention to the upper bound on element size. This appears to be an implementation error that we could repair either using the upper bound as hinted in the comments or entirely removing it from the code. However, since we desire to self-check the original implementation, we choose to set this parameter to MAXINT. This decision ensures that the behaviors of the constructors of the direct and by-hand implementation will be the same. The implications of this situation are further discussed in section 2.2.3.

The Direct Implementation

In the self-checking approach proposed in section 3, the direct implementation can be generated automatically. Here we show a design by displaying the code that could be generated from the intset axioms. In the C++ direct implementation a user-defined sort (of type absset) has a data structure that is a pointer to a discriminated union. The discriminant values are tokens whose values stand for constructors of the sort. Each arm of the union is a C++ struct whose components represent the arguments of the constructor associated with the discriminant. Dynamic polymorphism would be a more elegant alternative, but less portable to other languages. In the example:

```
typedef int elem;               // kind of generic
enum tag { EMPTY, INSERT };     // tokens for the discriminants

struct set_node {
  tag t;                        // constructor discriminant
  union {
    struct {
      int m;
      int r; } _0;              // arm associated with ‘‘empty’’
    struct {
      elem e;
      set_node* s; } _1;        // arm associated with ‘‘insert’’
  };
  set_node(int m, int r)        { t = EMPTY;  _0.m = m; _0.r = r; }
  set_node(elem e, set_node* s) { t = INSERT; _1.e = e; _1.s = s; }
};
typedef set_node* absset;

// Simple macro definitions to improve readability

#define tag_of(w)       (w->t)
#define maxsize_of(w)   (w->_0.m)
#define range_of(w)     (w->_0.r)
#define elem_of(w)      (w->_1.e)
#define set_of(w)       (w->_1.s)

// Declare a function for each signature symbol

extern absset empty(int m, int r);
extern absset insert(elem e, absset s);
extern int cardinality(absset s);
extern int maxsize(absset s);
extern bool member(elem e, absset s);
// equality-test function
extern bool equal(absset s1, absset s2);  // normal-form (syntactic) equality
```

Constructors and operations are implemented by functions without side effects. The execution of a function implementing a constructor dynamically allocates its associated union and returns a pointer to it. Each function implementing a non-constructor consists of a nest of cases whose labels correspond to the patterns in the rewrite rules. Rule conditions are implemented by conditional statements. Since both the patterns and the conditions are mutually exclusive the order of execution may affect the efficiency, but not the result, of a computation. The completeness of the patterns implies that the execution of a function implementing an operation is bound to find a matching rule and eventually to execute a call which represents the rule right side. Except for the case of “?” the execution of this call generates a finite tree of calls whose leaves are always calls to constructor functions and consequently an abstract representation of a sort instance is always returned. We translate the condition of a rule with “?” as the right side by means of a macro `exception` very similar to the macro `assert` provided by the GNU C++ compiler, which we use in our project.

```
#define exception(ex) \
  if (ex) { \
      cerr << "Exception ‘" << #ex \
           << form("’ at line %d of ‘%s’.\n", __LINE__, __FILE__); \
      abort (); \
  }
```

Examples of the direct implementation of constructor functions and an operation function follow.

```
absset empty(int m, int r)
{
  // empty(M,R) -> ? :- M<1 or R<M
  exception(m < 1);
  exception(r < m);
  absset s = new set_node(m,r);
  return s;
}

absset insert(elem e, absset s)
{
  absset h;
  switch tag_of(s) {
    case EMPTY:
      // insert(E,empty(_,R)) -> ? :- E > R
      exception(e > range_of(s));
      h = new set_node(e,s);
      break;
    case INSERT:
      ...
  }
  return h;
}

int cardinality(absset s)
{
  switch tag_of(s) {
```

```
  case EMPTY:  return 0;
  case INSERT: if (member(elem_of(s),set_of(s))) return cardinality(set_of(s));
               return (1 + cardinality(set_of(s)));
  }
}
```

2.2.3 Executing the Small Integer Set

The execution of the self-checking implementation of intset raises some interesting issues about the by-hand implementation in [56]. Although the documentation in the code seems to require an upper bound for the value of an element, this constraint is not enforced in the by-hand implementation. The self-checking implementation detects the problem during testing and issues the following warning:

```
Exception ‘e > range_of(s)’ at line 41 of ‘absset.C’.
```

The message “e > range_of(s)” is the textual code appearing in an exception macro in the direct implementation of insert. As the comment there indicates, the exception flags a violated condition in the axiom:

```
insert(E,empty(_,R)) -> ? :- E > R
```

This problem is undetected during the same test of the code in [56].

The direct implementation includes the operation cardinality that has no corresponding member function in the by-hand implementation of [56]. A naive programmer might add this observer function by returning the value of cursize, the counter of elements stored in the array that represents a set. However, the self-checking implementation would report a failure of such a cardinality on any test where the by-hand implementation creates a set with insert of duplicate elements. What the naive programmer missed is that the by-hand implementation in fact stores duplicates in its array. The specification we wrote for cardinality does not have this unexpected behavior, and would thus catch the mistake in the naive cardinality implementation.

The small-integer-set example illustrates the benefits gained from a formal specification. Direct implementation of the specification provides a careful check on a by-hand implementation, allowing self-checking of test values. Of course, it requires additional effort to write the specification; it can be argued, however, that without a formal specification, correct code is impossible to write. We have seen two examples of this in a well understood class of a textbook example.

2.2.4 An example of self-checking

An example will make clear the way in which the explicit representation function allows the results computed by the by-hand implementation to be checked against those specified by the direct implementation. Consider a previously created intset containing elements 1 and 5. Perhaps this set was created with the C++ code:

```
intset Ex(6,MAXINT);
...

Ex.insert(5);
Ex.insert(1);
```

Then it has already been checked that the state defined by the instance variables, including the concrete array in which the first two elements are 1 and 5, properly correspond to the term

insert(1,insert(5,empty(6,MAXINT))),

which is the normal form assigned to this state by the representation function.

$$\begin{tikzcd}
\mathit{insert(1,insert(5,empty(6,MAXINT)))} \arrow[r, "\mathit{insert}\ 2"] & \mathit{insert(1,insert(2,insert(5,empty(6,MAXINT))))} \\
\begin{array}{c}\texttt{cursize} = 2 \\ \texttt{maxsize} = 6 \\ \texttt{*x} = \begin{array}{|c|c|c|c|c|c|}\hline 1 & 5 & & & & \\ \hline\end{array}\end{array} \arrow[u, "\texttt{concr2abstr}"] \arrow[r, "\texttt{Ex.insert(2)}"] & \begin{array}{c}\texttt{cursize} = 3 \\ \texttt{maxsize} = 6 \\ \texttt{*x} = \begin{array}{|c|c|c|c|c|c|}\hline 1 & 2 & 5 & & & \\ \hline\end{array}\end{array} \arrow[u, "\texttt{concr2abstr}"]
\end{tikzcd}$$

Figure 3: Commuting diagram for `insert`ing 2 into the `Ex intset`.

Now suppose that the element 2 is added to this set, perhaps by the C++ call `Ex.insert(2)`. Figure 3 shows the particular case of the commuting diagram that checks this computation. At the lower level are the instance variables that comprise the concrete state, with the initial value at the left. The member function `insert` in the by-hand implementation transforms these variables as shown when called with arguments `Ex` and `2`. At the upper level are the corresponding abstract values, transformed by rewriting in the direct implementation. The explicit representation mapping `concr2abstr()` connects the two levels. From the concrete instance variables it constructs the abstract values, and the comparison that shows the computation to be correct occurs when the abstract value

insert(1,insert(2,insert(5,empty(6,MAXINT))))

is obtained by the two paths around the diagram starting at the lower left and ending at the upper right.

Of course, what is actually compared in the self-checking is not "abstract values," but bit patterns of a computer state, a state created by the compiled C++ program for the direct implementation. However, these states are so transparently like the structure of the abstract terms, as words in a word algebra, that it is obvious that they properly correspond. It is impossible to do better than this in any mechanical way. Mechanical comparisons must be done on computer states, not the true abstractions that exist only in a mathematical universe, and the best we can do is to make the states be simple, faithful mirrors of this universe.

3 A Proposed Automatic Testing System

The example of section 2.2 begins with two human-created objects: a specification and a by-hand implementation. We constructed the self-checking implementation from these by adding a few lines to the by-hand implementation, lines that call a direct implementation of the specification. We now consider how to automate the construction of these additional elements in the self-checking implementation. Mechanical creation of the self-checking implementation helps to justify the extra effort needed to create independent specifications and by-hand implementations.

3.1 Automating the Direct Implementation

The direct implementation of the specification is an idea presented and analyzed in [31]. The specification must be in the form of a term rewriting system. The implementation represents terms *directly*, i.e., using term independent structures of the implementation language, e.g., linked, dynamic structures. Computations of the direct implementation are performed by rewriting terms to their normal forms. Implementations of this kind are numerous and often add extra features to rewriting. For example, the *Equational Interpreter* [51] adds total laziness, OBJ3 [28] adds a sophisticated module system, and *SbReve* [2] adds a Knuth-Bendix completion procedure. These are stand-alone systems; in contrast, our direct implementation must appear as a block of C++ code to be integrated with the by-hand implementation. We also extend the treatment of [31] to conditional rules. Conditions add expressive power to the specification language without adding substantial complications to the direct implementation.

A data representation and the implementation of rewrite rules have been illustrated in section 2.2.2, and it is not difficult to "compile" the appropriate C++ code from the specification using compiler-compiler techniques, such as those of the Unix system [40, 44] or some other environment, e.g., *Standard ML* of New Jersey [6, 7]. The most difficult part of the compilation will be the "semantic" processing to guarantee that the specification rewrite rules possess the termination and confluence properties that make the direct implementation work. This could be eased by the syntax of the specification language. For example, by expressing in an **if ... then ... else** form the conditions of rules with overlapping left-hand sides, the instances of overlaps could be reduced or eliminated altogether.

The direct implementation discussed in section 2.2.2 uses a functional-like style instead of an object-oriented one. "Functional" code is easier to understand and natural for this application, since abstract objects have no internal state; and the encapsulation protection offered by a class is wasted in this case, since the direct-implementation code is created by a compiler and not for human use. The appendix describes a larger experiment in Java, a language that does not allow global functions, where the direct implementation of the specification is in a more "object-oriented" style.

3.2 Automating Calls on the Direct Implementation

The additional statements that must be added to the by-hand implementation are few, and present no difficulties. One way to automatically create them is to write a preprocessor from C++ into C++. Using a C++ grammar that omits most of the language's detail, with a compiler compiler that simply copies most code through directly, is one way to write the preprocessor quickly [10]. A second idea takes advantage of the existence of the parser in an existing C++ compiler. It is very easy to modify the code generator to insert object code for the necessary calls [34]. These ideas converge if the C++ compiler is itself more of a preprocessor (into C, say) than a true compiler. In the easiest case, such a preprocessor might be itself written using a compiler compiler.

There are a number of technical problems in modifying the by-hand implementation. For example, the abstract and concrete worlds share built-in types like Boolean, and for operations returning these types the representation function is identity. Thus the machinery of `abstract` and `verify` is not needed, and the inserted calls take a simpler form. A slightly more difficult problem arises for by-hand implementations that are not functional, such as `insert`. The usual implementation uses update in place, as in [56], so the abstract operation has a different arity than the concrete. Thus slightly different code is required:

```
void intset::insert(int t)
// Code from Stroustrup, section 3.2
{
  if (++cursize > maxsize) error("too many elements");
  int i = cursize-1;
  x[i] = t;

  while (i>0 && x[i-1]>x[i]) {
    int w = x[i];       // swap x[i] and [i-1]
    x[i] = x[i-1];
    x[i-1] = w;
    i--;
  }
  // Self-checking added
  abstract = ::insert(t,abstract);
  verify;
}
```

3.3 The Representation Function

There remains only the representation function that maps between the concrete and abstract domains, the function named `concr2abstr` in section 2.2.2. There seems to be no way that essential parts of this function can be automated. The correspondence between concrete and abstract objects is a primary design decision made early-on in the by-hand implementation, and the designer is not constrained in its choice. Furthermore, it is crucial to the proper working of the approach we propose that the representation correctly capture the link between concrete and abstract.

It can be argued that the extra programming required to code the representation function is a blessing in disguise. Unless the programmer has a detailed and accurate idea of this function, it is impossible to write correct functions that implement the specification's operations. What better way to force this understanding than to insist that it be put into code? What better way to protect against changes that are inconsistent with the representation than to make use of its code? (Both of these issues arose in the example above, section 2.2.3.) There is even an answer to the possibility that an incorrect representation function will trivialize self checking. Programmers are more likely to err in the direction of misguided elaboration than toward trivial simplicity. The more baroque a representation is, the less likely it is to conceal its faults; putting in too much will lead to our approach reporting ersatz failures, not to false success.

It has been suggested [46] that very often the representation function is structurally similar to a routine that pretty-prints a class value from its internal representation to human-readable form. This insight again underscores how easy it is to code the representation function, and how essential its capture is.

3.4 System Overview

Figure 4 shows the self-checking implementation that would result from the example in section 2.2. The direct implementation is invoked from the by-hand implementation by additional code that computes a term (`abstract`) in the data structure of the direct implementation, then applies the representation function `concr2abstr` to map the implementation state to a term, and compares these (`verify` macro).

Representation invariants [35], are generally necessary for proving the correctness of an implementation of a data type. However, they play a less significant role for our more modest goal—neither the truth nor the falsity of a supposed invariant assertion provides definite information on the correctness of a particular execution of an implementation.

Fundamental invariant assertions of a representation, when they can be identified, are shorthands for describing rewrite axioms with exceptions. One can think of checking the representation invariant before invoking every operator, or after every operator, or both. For example, the invariant assertion of class `intset`

$$1 \leq \textit{set maximum size} \leq \textit{elements upper bound}$$

directly leads to the first axiom of its specification.

4 Relation to Previous Work

Previous attempts to link formal specification to ADT implementation have taken a variety of forms.

Proof systems. The correctness of data type representation is proved using diagrams such as that presented in figure 2. Hoare [35] shows that the existence of a *representation mapping* R that makes the diagram commutative is proof of the implementation correctness. This mapping is somewhat "retrospective," since the concrete implementation originates from the abstract world which is the formalization of intuitive concepts.

Executable specifications. A specification is a non-procedural description of a computation. In an algebraic specification this description takes the form of a set of equations. When equations are given an orientation, i.e., are transformed into *rewrite rules*, the resulting structure, called a *rewrite system* [41], allows us to "compute." An elementary step of computation consists in rewriting some subterm of an expression by means of a rule. A computation is a sequence of elementary steps. Often two fundamental properties are required: *termination*, i.e., any computation sequence ends up in some element which cannot be further rewritten [18], and *confluence*, i.e., the choice of which term to rewrite in an expression does not affect the result

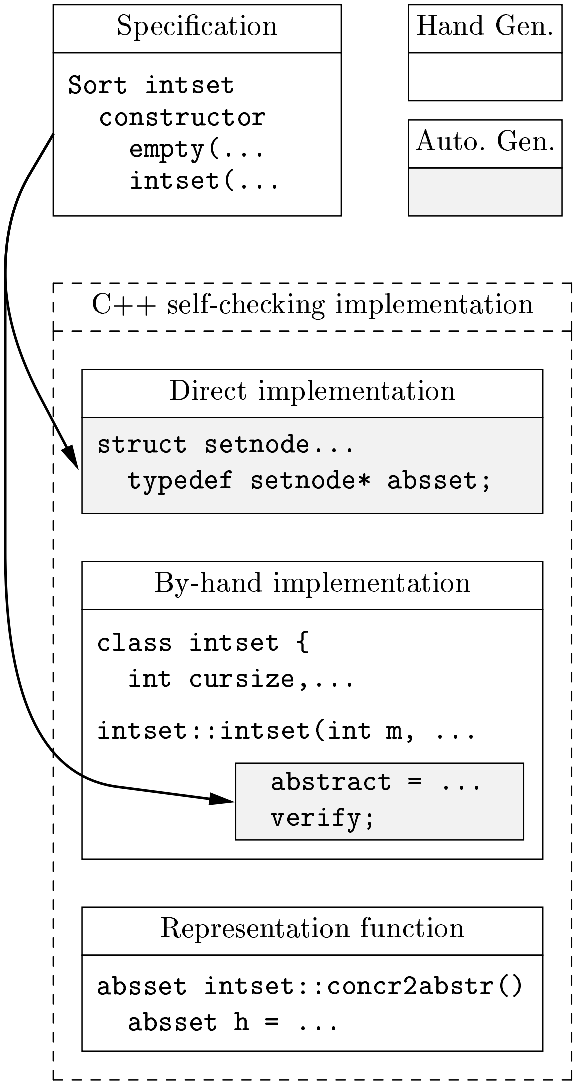

Figure 4: Construction of the self-checking implementation.

of the computation [39]. Rewrite systems with these properties are the model of computation underlying programming languages such as O'Donnell's *Equational Interpreter* [51] and Obj [28].

Automatic programming. If ADT specifications are viewed as a very-high-level programming language (and the executable nature of axiomatic specifications supports this view), then there is no need to write an implementation at all. The specification when executed *is* the implementation. Thus questions of correctness do not arise, and the only difficulty lies in improving the efficiency of execution. Antoy et al. [3, 5] investigate specification translation into a number of different languages. Volpano [57] proposes to "compile" the specification into an imperative language like C. Using ideas from functional languages like ML, he is able to effect this compilation, although in some cases the efficiency does not approach what would be obtained in an implementation by hand.

Testing systems. There appear to be three distinct threads in the attempt to use ADTs with tests.

First, any proof technique can be used to instrument implementation code with run-time assertions, which check test instances of proof assertions that could not be established by the theorem prover. The GYPSY system [1] uses this technique.

Second, ADT specifications can be used to formalize the generation of specification-based tests for an ADT implementation. Gerhart [25] describes a logic-programming approach to generate test points according to any scheme the tester imposes. In a slightly different approach, an ESPRIT project automatically generates tests based on traces (operation sequences) without direction by the tester [14, 15, 24].

Third, the DAISTS system [22] attempts to check consistency of an implementation and algebraic ADT specification, by executing the concrete code corresponding to the two sides of a specification axiom, and comparing the results with an implementation-supplied equality function.

Anna [47] specifications for Ada are intended to cut across these categories, but work has progressed less far than in the more specialized projects cited above.

Our approach might be described in these terms as a combination of a proof system and a testing system. In contrast to the executable specification approach, we consider both formal specification and independently devised implementation. (Perhaps both are derived from a common intuitive description.) In contrast to the automatic programming approach, the implementation code is not guaranteed to be correct by transformational origin; indeed, the implementation may be full of the tricks that are the death of formal proof (but essential in practice for efficiency).

The view that specifications are their own implementation is attractive; for one thing, it cuts the work of specification and implementation in half. However, its drawback is that it merely moves the correctness problem up a level. However carefully a specification is devised, it may fail to capture the intuitive ideas of problem solution that led to it, and that intuition necessarily exists on a plane inaccessible to formal methods. Hence it may be wise to duplicate the work of problem solution in very different specifications and implementations, both drawing on the intuitive problem model. One may then hope that where the model is unclear or faulty, the independent attempts to capture it formally will differ, and precise correctness methods will detect the difficulty and lead to correction of the model.

Unlike those who use proof systems, we do not attempt to verify the implementation, but only to check it for particular cases. (The return for this drastic restriction is a large improvement in the success rate of the process, and a lowering of the skill level needed to use it.) Ours is a testing approach using some proof techniques, whose nearest precursor is DAISTS [22]. Unlike DAISTS, however, our test for equality of ADT values is conducted in the abstract domain (hence the proof techniques) rather than in the concrete domain. We thus avoid both practical and theoretical deficiencies that could falsify a DAISTS success, yet do not pay an efficiency penalty, because we use rewriting for the abstract proofs, with an explicit mapping from the concrete to abstract domain. This mapping and its expression as part of the implementation are our main contribution. The extra programming we require for the representation function corresponds to the need for the DAISTS programmer to explicitly code a concrete equality function, but is easier and more natural.

Our approach can also be viewed as checking runtime behavior using code assertions. Unlike ad hoc systems [48] or proof-based systems such as GYPSY [1], however, these code assertions are not directly written by the user, even in conjunction with a theorem prover. Rather, they are automatically generated from the specification and inserted in the code where appropriate. If an execution of the code deviates from its specification, these assertions are guaranteed to detect it.

We do not generate test data, nor judge the adequacy of test data, but any scheme that does generate tests [37, 38] or measure test quality [17, 50] can be used with our approach supplying the test oracle, a facility missing from most testing systems with a few noticeable exceptions [58]. Frankl and Doong [20] describe a system that uses rewriting to obtain one (abstract) test case from another, so that the results of an implementation can be compared on these cases. Sankar [55] uses a much more powerful rewriting theorem prover to attempt to prove abstract equality between all terms an implementation generates. Antoy and Gannon [4] use rewrite systems similar to ours to prove the correctness of loops and subtypes with the help of a theorem prover. All these systems are less straightforward than ours, because they lack the explicit representation function and/or the specification restrictions needed to guarantee rewriting termination.

5 Discussion

Compared to automatic proof schemes for ADTs, and to automatic programming of efficient programs from formal specifications, the goal for our testing approach is modest. We imagine no more than an automatically generated, perfect set of run-time assertions, which make it impossible for an execution of the by-hand implementation to silently disagree with the specification. We can attain the limited goal where more ambitious ones present formidable problems, but is it worthwhile? In this section we try to answer that question in the affirmative.

5.1 The Need for Test Oracles

The testing literature almost universally assumes that test output is examined for correctness; and, it almost universally fails to say how this can be done. Furthermore, research examples [53] and empirical studies [11] alike show that it is common for testers to have failures in their hands, yet ignore them. Thus the "effective oracle problem"—the difficulty of mechanically judging if a program's output meets specifications—is an important one. It assumes extra importance for random testing. Recent work suggests that true random testing based on a valid operational profile is essential, and that confidence in such tests requires a vast number of test points [33]. The adequacy criteria in widespread use require hundreds of points; adequate random testing requires millions. Such tests are flatly impractical without an effective oracle.

Given the oracle, random testing is doubly attractive, however. Not only is it theoretically valid, able to provide a true estimate of reliability by definition [32], but it approximates the ideal of a "completely automatic" test. The random inputs can be selected mechanically, and with a means of mechanically examining outputs, a test suite can be run without the need for human intervention.

5.2 Multi-version Specification/Programming

When a specification is viewed as a program in a very high level language, yet no general algorithm exists for compiling that language into efficient code, there is still a place for by-hand implementation. In this view, the development process is directed. First comes a requirements phase in which the developers, in communication with the end users, attempt to create a formal specification that captures the necessary intuitive problem solution. In this process the prototyping aspect (see section 5.3) of the specification formalism is of great importance. Next, the specification is efficiently implemented, automatically where possible, by hand otherwise. Formal methods are used to show that this implementation is correct. In practice, we believe that there will always be a need for by-hand implementations, and that general methods of proof will always need to be supplemented by tests. The approach we have proposed can automate the testing process efficiently.

One can argue that when development proceeds from requirements to formal specification to by-hand implementation, the declarative form of specification is not the best. Rather, a form of specification much closer to the ultimate imperative implementation language is called for [59]. The advantages are twofold: first, such procedural specifications are easier to write; and second, many of the detailed problems of an imperative-program solution must be addressed in the prototype, so that the subsequent by-hand implementation is easier to write, and less prone to introduce subtle disagreements with the specification.

However, there is a rather different view of specification/implementation in program development. In this view, both specification and implementation are imperfect reflections of intuitive requirements for problem solution. This view is particularly appropriate in safety-critical applications. In an attempt to provide software fault tolerance, the technique of *multi-version programming* (MVP) has been suggested [9]. However, it has been observed [42] that so-called "common-mode failures" are more frequent than might be expected—working from the same (informal) specification, independent programming teams make mistakes that lead to coincident wrong results. The proposed solution is *design diversity*, that is, programs that differ so radically that they are truly independent. A recent study [45] casts doubt on the whole idea of MVP, and in contrast suggests that internal self-checking is more valuable, particularly when the checking involves the details of internal program states.

The approach we propose fits the needs of safety-critical applications very well. It is the ultimate in self-checking code, and the checks are applied to internal data-structure states, through the explicit representation function that maps those states into the direct implementation. At the same time, because the direct implementation is executed, a self-checking implementation can be viewed as a two-version programming package with, in most cases, a distinctive design diversity. The declarative nature of the axiomatic specification, and its direct execution by rewriting, should make common-mode failure with a conventional by-hand implementation unlikely.

5.3 Rapid Prototyping

Previous systems implementing specifications directly, such as OBJ3 [28], have been designed so that specifications can be executed as prototypes, to allow potential users to interact with software before a complete development effort freezes the wrong design. Our direct implementation adds a new dimension to this idea. The by-hand implementation and the direct one implement the same specification completely and independently. In our software development approach we use the former for production, both for testing, and the latter for prototyping. Our two implementations coexist in the same environment, that of the final product. In earlier systems prototypes are confined to unusual software and/or hardware platforms (e.g., Obj3 lives inside Common Lisp). Our prototype and production modules interact in very similar ways with the rest of the system. From an external point of view, the only difference between the two versions of an operation is that the direct implementation is side-effect free, while the by-hand implementation, for efficiency reasons, might not be. This gap can be filled by a trivial interface limited to renaming operations and rearranging parameters and modes of function declarations.

6 Summary and Future Work

We have proposed a modest testing approach for modules using an algebraic specification executable as rewrite rules. The programmer must write a specification, C++ code to implement it, and a representation function relating implementation data structures to abstract terms. From these three elements a self-checking implementation can be constructed automatically, in which the specification serves as a test oracle. The self-checking implementation can be viewed as a vehicle for testing only, or as a special kind of two-version programming approach with exceptional design diversity.

We are pursuing two quite different goals for the future. First, we are investigating more expressive languages for the formal specification component of our approach, and by-hand implementation languages, such as Java, that are simpler than C++. Second, we want to use these ideas in a practical setting, to learn more about the difficulty of writing specifications, and the value of a test oracle. The ideal testbed is an industrial user of object-oriented design with a central group responsible for developing highly reliable support software that other developers use. In such a group, it should be worthwhile putting the extra effort into specification, in return for better testing and reliability.

Appendix

This appendix describes the application of self-checking to a module of realistic size and complexity. Java package *util* contains a class, called *Vector*, that efficiently implements a data type (defined below). In the following we refer to version 1.36 of the code that is distributed with version 1.1 of the Java Development Kit. Class *Vector* is the largest class abstracting a container and one of the largest classes of the entire package. The size of class *Vector* is representative of Java's approach to the implementation of ADTs and we choose it to assess how our technique scales to industrial software of a practical size.

A specification will be created for *Vector* starting with the common intuition about this data type, and fitting to the details of the actual code in package *util* only as the last step. We produce an initial simple specification from our intuition of a vector. We assume that some concerns about the efficiency of the implementation suggest a slightly different abstraction. Thus we produce a second specification based on the first one, where some requirements of an efficient implementation creep in. Finally, we retrofit our second specification to class *Vector* of Java package *util*. This stepwise refinement of specifications promotes understanding and increases our confidence that our second specification is still an adequate formalization of our intuition.

A vector is an indexed collection similar to a built-in array of most programming languages. It differs from a typical array in that it provides operations to insert or remove elements at a given index. Consequently a vector can be extended or contracted at run time. Formally, a vector is a dynamic mapping from a totally ordered set of indices, which without loss of generality we take to be the set of natural numbers, to a set of elements that we refer to as *objects*.

Sort *vector* has two constructors: *empty*, which creates an empty mapping, and *map*, which maps a natural number to some object. We specify an operation to *remove* an index's mapping (the insertion is similar), and a handful of typical observer operations. The normal forms of sort *vector* are ordered sequences of *map* operations. Attempting to compute the index or the object of a non-existing mapping produces an error. Both ground confluence and termination of this specification are easy to verify.

The only recursive constructor in the specification of *vector* is *map*. The recursive argument of *map* is the last one. Thus, the symbol "!" in the right-hand side of a rule with left-hand side of the form $f(\cdots map(i,o,v)\cdots)$ stands for $f(\cdots v \cdots)$. The right-hand side is obtained from the left-hand side by replacing the subterm $map(i,o,v)$ with v; every other portion of the left-hand side is unchanged. For example, without the recursive reduction notation the second axiom of *size* would be written

$$size(map(\_,\_,V)) \rightarrow succ(size(V))$$

The recursive reduction notation better expresses that a computation depends on the structure of a term rather than its value and trivializes the proof of termination of rewriting which is essential to the completeness of the specification.

```
sort vector
  constructor
    empty
    map(natural,object,vector)
  axiom
    map(I,O,map(I',O',V)) -> map(I',O',map(I,O,V)) <= I < I'
    map(I,O,map(I',_,V)) -> map(I,O,V)             <= I = I'

operation remove(natural,vector) -> vector
  axiom
    remove(_,empty) -> empty
    remove(I',map(I,O,V)) -> if I > I' then map(I-1,O,!) else
                             if I = I' then V else map(I,O,!)

operation size(vector) -> natural
  axiom
    size(empty) -> 0
    size(map(_,_,_)) -> succ(!)

operation is_empty(vector) -> boolean
  axiom
    is_empty(V) -> size(V) = 0

operation is_mapped(natural,vector) -> boolean
  axiom
    is_mapped(empty) -> false
    is_mapped(I',map(I,_,_)) -> I' = I or !

operation element_at(natural,vector) -> object
  axiom
    element_at(_,empty) -> ?
    element_at(I',map(I,O,_)) -> if I' = I then O else !

operation contains(object,vector) -> boolean
  axiom
    contains(empty) -> false
    contains(O',map(_,O,_)) -> O' = O or !

operation index_of(object,vector) -> natural
  axiom
    index_of(_,empty) -> ?
    index_of(O',map(I,O,_)) -> if O' = O then I else !
```

For example, the normal form of the vector mapping index 0 to object E_0 and index 1 to object E_1 is $map(1, E_1, map(0, E_0, empty))$ regardless of the order in which the mapping operations are actually performed.

The Java implementation of class *Vector* represents the mapping using an array, which we refer to as *underlying.* This representation makes the retrieval of the object mapped by an index efficient. When a mapping is created for an index outside the bounds of the underlying array, a new larger array is allocated and the content of the old array is copied into the new one. This operation is expensive. Class *Vector* offers some control to reduce the number of times that this operation is performed. The class allows the programmer access to both the *initial capacity* of the underlying array and the *capacity increment* to be used when the array overflows and a new larger array must be allocated. The class also allows access to the *size* of the vector. Indices in the range 0 though $size - 1$ that are not mapped explicitly are mapped by default to the distinguished object *null.* Indices outside this range are inaccessible.

The specification of Java's class *Vector* is of course larger than our initial specification, because class *Vector* has more methods, but conceptually it is similar. We briefly outline the most significant differences. *Initial capacity* and *capacity increment* are specified as arguments of the *empty* constructor of the sort *vector.* Similar parameters were used in the specification of the C++ implementation of a bounded set discussed in section 2.2.1. *Size* is initialized by constructor *empty*, reset by method *setSize* and updated by removals and insertions of objects. The specification does not of course allocate an array and makes a limited use of both *initial capacity* and *capacity increment.* Some methods of the class return integer value -1 as the index of objects that are not in a vector. Thus our new specification replaces naturals with integers for the indices. Interestingly enough, class *Vector* provides a method, *toString*, to print a human readable representation of a vector. As noted in section 3.3, this method suggests how to code the representation mapping needed by the self-checking implementation.

In Java the direct implementation of the specification differs from C++, since Java forbids both global functions and unions. We translate a sort into an *abstract* class with no instance variables. For example, sort *vector* is translated into

```
abstract class absVector {  ...  }
```

A term of sort *vector* is directly implemented by an instance of a subclass of *absVector*. There are as many subclasses as there are constructors. For example, class *absVector* has two subclasses, one for constructor *empty* and another for constructor *map*. The subclass directly implementing constructor *empty* is shown below. The subclass directly implementing constructor *map* is similar.

```
class absEmpty extends absVector {
  int size, increment, capacity;
  absEmpty (int S, int I, int C) { size=S; increment=I; capacity=C; }
}
```

Defined operations are *static* methods of abstract classes. This is a natural choice since a defined operation has no side effects and the abstract class it belongs to has no state. The pattern matching performed by most operations is implemented using the strong run-time type information facilities available in Java, e.g., in class *absVector* we find

```
static final int capacity (absVector v) {  // compute the capacity of a vector
  if     (v instanceof absEmpty) return ((absEmpty) v).capacity;
  else /* v instanceof absMap */ return capacity(((absMap) v).vector);
}
```

Certain methods of class *Vector* throw exceptions when called with certain arguments. When a vector throws an exception, its state is unchanged. Because of our backward approach to retrofit the specification to the code, we can simply ignore exceptions in the specification. In fact, self-checking does not occur when an exception is thrown, since the method throwing the exception does not reach its normal exit point. If self-checking were performed, the specification would simply have to throw an exception as well. In a forward approach to specification and implementation, exception handling requires only a modest increase in complexity. Self-checking the exceptional termination of a method entails verifying that the specification terminates exceptionally, and as in the case of normal termination, that the concrete state is the representation of the abstract one.

We self-check Java class *Vector* just as we did the C++ class *intset* in section 2.2. After designing the specification we produce its direct implementation. This step, which would take negligible time for a compiler, takes several hours by-hand and inadvertently we introduced a small mistake. Coding the representation function and binding together the by-hand and the direct implementations is straightforward.

We conducted three separate sets of tests of class *Vector*. The first consisted of a totally random test. It consisted of 10^6 calls to the public operations (constructors and methods) of the class. Each operation had the same probability of being called. The arguments, if any, of each operation were randomly generated within predefined ranges. This test exposed the fault we inadvertently introduced in the direct implementation. We fixed the fault and completed the test in a total execution time of about 20 min on a modern laptop.

The second test set corrected a deficiency of the totally random test. In the previous test every operation of the class had the same probability of being executed. A consequence was that a vector had a short life span, since new vectors replaced existing vectors rather frequently. This did not adequately test the overflow of a vector and its reallocation. Thus, in the second set of tests we avoided new constructions and the operation that remove all the elements of a vector. A sequence of 10^4 calls allowed us to self-check a large number of reallocations of a vector. We tried various combinations of ranges for the randomly generated indices and objects.

Since the above tests produced no errors, we designed a third set of tests where we purposedly introduced errors in the code of class *Vector*. For example, class *Vector* provides a method, *lastIndexOf*, to return the maximum index of a given object. This method scans the underlying array from the maximum index down to zero with the following *for* loop

```
for (int i = index ; i >= 0 ; i--) ...
```

We introduced an error in the termination condition of the loop.

```
for (int i = index ; i > 0 ; i--) ...
```

When we tested the incorrect program the randomly choosen values of the vector's *size* and *increment* were 14 and 13 respectively. The test used 100 different objects. The self-checking implementation detected the error after about 27,500 random calls to the vector's public operations. By the time the error was detected, the size of the vector had grown to 295.

The code of our experiment is available on-line at:

`http://www.cs.pdx.edu/~antoy/selfcheck/index.html`

References

[1] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells. Gypsy: A language for specification and implementation of verifiable programs. In *Language Design for Reliable Software*, pages 1–10, Raleigh, NC, 1977.

[2] S. Anantharaman, J. Hsiang, and J. Mzali. *SbReve2*: A term rewriting laboratory with (AC-)unfailing completion. In *RTA'89*, pages 533–537. Springer-Verlag, 1989.

[3] 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.

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

[5] S. Antoy, P. Forcheri J. Gannon, and M.T. Molfino. Equational specifications: Design, implementation, and reasoning. In A. Miola and M. Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, pages 126–144. Springer, 1996. Texts and monographs in symbolic computation.

[6] A. W. Appel, J. S. Mattson, and D. R. Tarditi. *A lexical analyzer generator for Standard ML.* Princeton University, December 1989.

[7] A. W. Appel, J. S. Mattson, and D. R. Tarditi. *ML-Yacc, version 2.0.* School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1990.

[8] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, Reading, MA, 1996.

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

[10] A. Babbitt, S. Powell, and R. Hamlet. Prototype testing tools. Technical Report TR90-8, Portland State University, Portland OR, 1990.

[11] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. on Soft. Eng.*, 13:1278–1296, 1987.

[12] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* Addison-Wesley, Wokingham, England, 1989.

[13] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella, editors. *Algebraic System Specification and Development.* Springer-verlag, 1991. Lect. Note in Comp. Sci., Vol. 501.

[14] L. Bouge, N. Choquet, L. Fribourg, and M-C Gaudel. Application of prolog to test sets generation for algebraic specifications. In *TAPSOFT*, pages 261–275. Springer-Verlag, 1985.

[15] N. Choquet. Test data generation using a Prolog with constraints. In *Workshop on Software Testing*, pages 132–141, Banff, Canada, 1986.

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

[17] M. D. Davis and E. J. Weyuker. A formal notion of program-based test data adequacy. *Information and Control*, 56:52–71, 1983.

[18] N. Dershowitz. Termination. In *RTA'85*, pages 180–223, Dijon, France, May 1985. Springer-Verlag.

[19] N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of conditional term rewrite systems. In *CTRS'87*, pages 31–44, Orsay, France, 1987. Springer-Verlag.

[20] R-K. Doong and P. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 165–177, Victoria, B.C., 1991.

[21] M. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[22] J. Gannon, R. Hamlet, and P. McMullin. Data abstraction implementation, specification, and testing. *ACM Trans. Prog. Lang. and Systems*, 3:211–223, 1981.

[23] J. D. Gannon, R. G. Hamlet, and H. D. Mills. Theory of modules. *IEEE Trans. on Soft. Eng.*, 13, 1987.

[24] M.-C. Gaudel and B. Marre. Generation of test data from algebraic specifications. In *Second Workshop on Software Testing, Verification, and Analysis*, pages 138–139, Banff, Canada, 1988.

[25] S. Gerhart. Test generation method using prolog. Technical Report TR 85-02, Wang Institute of Graduate Studies, 1985.

[26] J. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebras. In W. Brauer, editor, *CALP'85*. Springer-Verlag, 1985.

[27] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice-Hall, Englewood Cliff, NJ, 1978.

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

[29] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[30] J. V. Guttag, J. J. Horning, and J.M. Wing. The Larch family of specification languages. *IEEE Software*, pages 24–36, Sept. 1985.

[31] J.V. Guttag, E. Horowitz, and D. Musser. Abstract data types and software validation. *CACM*, 21:1048–1064, 1978.

[32] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.

[33] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.

[34] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. on Soft. Eng.*, 3:279–290, 1977.

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

[36] D. Hoffman. Hardware testing and software ICs. In *Pacific Northwest Software Quality Conference*, pages 234–244, Portland, OR, 1989.

[37] D. Hoffman and C. Brealey. Module test case generation. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 97–102, Key West, FL, 1989.

[38] W. Howden. Methodology for the generation of program test data. *IEEE Trans. Computers*, 24:554–559, 1975.

[39] G. Huet. Confluent reductions: Abstract properties and applications to term-rewriting systems. *JACM*, 27:797–821, 1980.

[40] S. C. Johnson. YACC: yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 388–400, New York, 1983. Holt, Reinhart, Winston.

[41] J. W. Klop. Term rewriting systems. Technical Report CS-R9073, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1990.

[42] 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.

[43] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press, New York, NY, 1970.

[44] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 353–387, New York, 1983. Holt, Reinhart, Winston.

[45] 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.

[46] R. London. Personal communication. February 1991.

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

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

[49] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, NY, 1987.

[50] S. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. on Soft. Eng.*, 14:250–256, 1988.

[51] M. J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.

[52] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM*, 15:1053–1058, 1972.

[53] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Trans. on Soft. Eng.*, 2:293–300, 1976.

[54] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Eng.*, 11:367–375, 1985.

[55] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 123–129, Victoria, B.C., 1991.

[56] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.

[57] D. M. Volpano and R. B. Kieburtz. Software templates. In *ICSE'85*, pages 55–60, London, UK, 1985.

[58] C. Wang and D. R. Musser. Dynamic verification of C++ generic algorithms. *IEEE Trans. Soft. Eng.*, 23(5):314–323, 1997.

[59] P. Zave. The operational versus the conventional approach to software development. *Comm. of the ACM*, 27:104–118, 1984.