6, depending upon the type of instruction currently being executed.

In all systems that use program counters, there must be a mechanism for initializing its value and for changing values at certain points in the program. This latter mechanism is a special instruction, usually called a *branch* or *jump*. There are two basic kinds of branch instructions—*unconditional branch* and *conditional branch*. The unconditional branch causes a new value to be placed in the program counter and hence defines the start of the location of a new sequence of instructions. A particularly important type of unconditional branch is the subroutine call, which additionally saves the current value of the program counter for later restoration. The conditional branch has a similar action except that it is dependent upon the state of certain data items. Thus, whether the next instruction will be simply the next instruction in the current sequence or the beginning of a new sequence will depend upon the result (e.g. positive or negative) of a preceding instruction

Michael J. Flynn

PROGRAM LIBRARY

See MATHEMATICAL SOFTWARE; and SOFTWARE LIBRARIES, NUMERICAL AND STATISTICAL.

PROGRAM SPECIFICATION

For articles on related subjects *see* ABSTRACT DATA TYPE; AUTOMATIC PROGRAMMING; FORMAL METHODS FOR COMPUTER SYSTEMS; INFORMATION HIDING; PROGRAM VERIFICATION; SOFTWARE ENGINEERING; SOFTWARE PROJECT MANAGEMENT; SOFTWARE PROTOTYPING; and SOFTWARE TESTING.

The term *program specification* may refer to:

1. A statement of *requirements* for a program.
2. An expression of a *design* for a program.
3. A formal statement of conditions against which the program can be *verified*.

Properties of Specifications

Whatever the kind of specification, there are several concerns:

1. *Consistency*—Is the specification logically satisfiable?
2. *Implementability*—Is the specification practically realizable?
3. *Completeness*—Does the specification capture the *full* intent of the specifier?
4. *Nonambiguity*—Does the specification capture the *precise* intent of the specifier?

Uses of Specifications

Specifications can be used in all phases of program development. In the *requirement analysis* phase, a specification helps crystallize the customer's possibly vague ideas and reveals contradictions, ambiguities, and incompleteness in the requirements. In *program design*, a specification captures precisely the interfaces between the modules of a program. Each interface specification provides the module's client the information needed to use the module without knowledge of its implementation, and simultaneously provides the module's implementer the information needed to create the module without knowledge of its clients. In *program verification*, a specification is the statement against which a program is proved correct. Verification is the process of showing the consistency between a program and its specification. In *program validation*, a specification can be used to generate test cases for black-box testing. Together with the program, it can be used for path testing, unit testing, and integration testing. Finally, a specification serves as a kind of *program documentation*, since it is an alternative, usually more abstract, description of a program's behavior.

For a more detailed discussion of formal specifications, see Wing (1990).

EXAMPLE

Consider the specification of a data abstraction for a *bag* (in the sense of a sack that holds inserted items). This example is taken from Guttag *et al.* (1985). Using the Larch specification languages, we divide the specification into two parts. The first part, called a *trait*, specifies state-independent properties of data accessed by programs; the second part, called an *interface*, specifies state-dependent behavior (e.g. side effects (*q.v.*) and exceptional termination of program modules). In the Larch approach, there is an interface specification for each programming language. For example, a Larch/Pascal interface specification describes the behavior of a Pascal (*q.v.*) program; it would look different from a Larch/C interface specification.

Fig. 1 presents a trait that is useful for describing values of multisets and is written in the style of algebraic specifications. A multiset is an unordered collection of items that may contain duplicates. A trait defines a set of function symbols and a set of equations that define the meaning of the function symbols. The equations determine an equivalence relation on terms written using the function symbols. The `generatedby` clause states that all multiset values can be represented by terms composed solely of the two function

```
MultiSet: trait
   introduces
      new: -> MSet
      insert: MSet,E -> MSet
      isEmpty: MSet -> Bool
      size: MSet -> Card
      count: MSet,E -> Card
      delete: MSet,E -> MSet
      numElements: MSet -> Card
    constrains MSet so that
      MSet generated by [new,insert]
      MSet partitioned by [count] for all [c: MSet, e, e1, e2:E]
         isEmpty(new) = true
         isEmpty(insert(c,e)) = false

         size(new) = 0
         size(insert(c,e)) = size(c) + 1

         count(new, e1) = 0
         count(insert(c,e1), e2) = count (c,e2) + (if e1 = e2 then 1 else 0)

         numElements(new) = 0
         numElements(insert(c,e)) = numElements(c) + (if count(c,e) > 0 then 0 else 1)

         delete(new, e1) = new
         delete(insert(c,e1), e2) = if e1 = e2 then c else insert (delete(c,e2),e1)
         implies converts [isEmpty,size,count,delete,numElements]
```

Figure 1. Specification of multiset values.

symbols, new and insert. This clause defines an inductive rule of inference and is useful for proving properties about all multiset values. The **partitioned by** clause adds more equivalences between terms. Intuitively, it states that two terms are equal if they cannot be distinguished by any of the functions listed in the clause. In the example, we could use this property to show that order of insertion of elements in a multiset does not matter (i.e. insertion is commutative). The **converts** clause is a way to state that this algebraic specification is sufficiently complete.

Fig. 2 gives a Larch/Pascal interface specification of a bag data abstraction. It introduces a type name, three procedures, and one function.

The body of each routine's specification places constraints on proper arguments for calls on the routine and defines the relevant aspects of the routine's behavior when it is properly called. It can be straightforwardly translated to a first-order predicate over two states by combining its three predicates into a single predicate of the form

```
requires predicate ⇒
 (modifies predicate & ensures predicate).
```

An omitted **requires** is interpreted as **true**.

In the body of a Larch/Pascal specification, as in Pascal, the name of a function stands for the value returned by that function. Formal parameters may appear unqualified or qualified. An unqualified formal parameter stands for the value of that parameter when the routine is called. A formal parameter qualified by prime ($'$), for example b', stands for the value of that formal parameter when the routine returns.

The values of variables on entry to and return from routines must be distinguished because Pascal is a language in which statements may alter memory. Since the function symbols in a Larch trait specification represent functions, this complication does not arise there, nor would it in an interface language for a functional programming ($q.v.$) language.

The **modifies** predicate is also related to the imperative nature of Pascal. The predicate **modifies at most** $[v_1, \ldots, v_n]$ asserts that the routine changes the value of no variable in the environment of the caller except possibly some subset of the variables denoted by the elements of $\{v_1, \ldots, v_n\}$. Notice that this predicate is really an assertion about all variables that do not appear in the list, not about those that do.

The **based on** clause associates the type *Bag* with the sort *MSet* that appears in trait *MultiSet*. This association means that Larch trait terms of algebraic sort *MSet* are used to represent Pascal values of type *Bag*. For example, the term "new" is used to represent the value that b is to have when *bagInit* returns. The **requires** clause of *bagAdd* states a precondition that is to be satisfied on each call. It reflects the specifier's concern with how this type can be implemented in Pascal. By putting a bound on the number of distinct elements in the *Bag*, the specification allows a fixed-size representation. It is quite natural for such considerations to surface in interface specifications; it would not be so natural for them to appear in traits.

```
type Bag exports bagInit,bagAdd,bagRemove,bagChoose
      based on sort MSet from MultiSet with [integer for E]
      procedure bagInit(var b: Bag)
            modifies at most [b]
            ensures b' = new

      procedure bagAdd (var b: Bag; e: integer)
            requires numElements(insert(b,e)) < = 100
            modifies at most [b]
            ensures b' = insert (b,e)

      procedure bagRemove (var b: Bag; e: integer)
            modifies at most [b]
            ensures b' = delete(b,e)

      function bagChoose (b: Bag; var e:integer): boolean
            modifies at most [e]
            ensures if ~isEmpty(b) then bagChoose & count(b,e') > 0
                        else ~bagChoose & modifies nothing
```

Figure 2. Interface specifications of a Larch/Pascal bag abstraction.

The most interesting routine is probably *bagChoose*. Its specification says that it must set *e* to some value in *b* (if *b* is not empty, where '~' denotes negation), but does not say which value. Moreover, it doesn't even require that different invocations of *bagChoose* with the same value produce the same result; in other words, the implementation may be *nondeterministic*. Our implementation is abstractly nondeterministic, even though it is a deterministic program (*see* Fig. 3). The value to which *e* is set depends on the order in which elements have been added to and removed from *b*, whereas this order does not affect *b*'s abstract value.

This interface specification has recorded a number of design decisions beyond those contained in the trait *MultiSet*. It says which routines must be implemented and, for each routine, it indicates both the condition that must hold at the point of call and the condition that must hold upon return. Thus, a contract that provides a *logical firewall* has been established between the implementers and the clients of type *Bag*. They can then proceed independently, relying only on the interface specification (*see* ABSTRACT DATA TYPE and INFORMATION HIDING).

The clients must establish the `requires` clause at each point of call. Having done that, they may presume the truth of the `ensures` clause on return, and that only variables in the `modifies at most` clause are changed. They need not be concerned with how this happens.

The implementers are entitled to presume the truth of the `requires` clause on entry. Given that, they must establish the `ensures` clause on return, while respecting the `modifies at most` clause.

Because the interface specification does not specify either the representation of the type or the algorithms in routines, yet another level of design is needed. Because this level is hidden from clients of the data type, the design may be changed without affecting their correctness.

The specification of each routine in an interface can be understood without reference to the specifications of other routines—unlike traits, in which the specification constrains the operators by giving relations among them. Of course, to understand the type itself, to reason about it, or to design an efficient representation for it, the specifications of all its routines must be taken into account.

To illustrate the relation between an interface specification and an implementation, we give a Pascal implementation of type *Bag* in Fig. 3. Neither the data structure chosen for the representation nor the program itself is very interesting. Both the abstraction function and the representation invariant are presented informally. If we had included a formal specification of the type used in the representation, we could have presented them using a program annotation language. Then they could be mechanically combined with the interface specifications already given to derive a concrete specification for each routine, which could then be verified separately.

For example, to show that the implementation of *bagAdd* satisfies its specification, one assumes the precondition, which says that there cannot be more than 100 distinct elements contained in the bag if we were to insert *e*. The implementation of *bagAdd* then either finds an index *lastEmpty* at which to insert a new distinct element *e* in the *elems* array (and sets the corresponding count for *e* to1) or finds the index *i* at which the *elems* array already stores *e* (and increments *e*'s count by 1). Notice that the implementation of *bagAdd* relies on the precondition since if we try to insert a new distinct element in a bag that already has 100 distinct elements then *lastEmpty* will be undefined

```
const MaxBagSize = 100;
type ElemVals = array [1..MaxBagSize] of integer;
   ElemCounts = array [1..MaxBagSize] of integer;
   Bag = record
            elems: ElemVals;
            counts: ElemCounts
            end;

{Abstraction function: the abstract bag is equivalent to the result of
inserting into the empty bag each integer in elems a number of times
equal to the corresponding number in counts.}

{Representation invariant: Each integer in counts is at least zero and
no integer appears in elems more than once associated with a positive
value in counts.}

procedure bagInit(var b: Bag);
   var i: 1..MaxBagSize;
   begin
      for i := 1 to MaxBagSize do b.counts[i] := 0
   end {bagInit};

procedure bagAdd(var b: Bag; e: integer);
   var i,lastEmpty: 1..MaxBagSize;
   begin
      i := 1;
      while (i < MaxBagSize) and (b.elems[i] <> e) do
         begin
            if b.counts[i] = 0 then lastEmpty := i;
            i := i + 1
         end;
      if b.elems[i] = e then b.counts[i] := b.counts[i] + 1 else
         begin
            if b.counts[i] = 0 then lastEmpty := i;
            b.elems[lastEmpty] := e;
            b.counts[lastEmpty] := 1
         end
   end {bagAdd};

procedure bagRemove (var b: Bag; e: integer);
   var i: 1..MaxBagSize;
   begin
      i := 1;
      while (not ((b.elems[i] = e) and (b.counts[i] > 0)) and (i < MaxBagSize)) do
         i := i + 1;
      if (b.elems[i] = e) and (b.counts[i] > 0) then
         b.counts [i] := b.counts [i] - 1
   end {bagRemove};

function bagChoose(b: Bag; var e: integer): boolean;
   var i: 1..MaxBagSize;
   begin
      i := 1;
      while (i < MaxBagSize) and (b.counts[i] = 0) do i := i + 1;
      if b.counts [i] = 0 then bagChoose := false else
         begin
            e := b.elems[i];
            bagChoose := true
         end
   end {bagChoose};
```

Figure 3. Pascal implementation of bag abstraction.

and an error presumably will occur in trying to access the *elems* array at an undefined index. Upon termination, *bagAdd* satisfies the postcondition of the specification, which says that the new bag value is the same as the old with the addition of the newly inserted element. Furthermore, we can check that upon inserting an element in a bag, its final (multiset) value satisfies the desired properties as specified in the *MultiSet* trait: inserting a new distinct element into a multiset increases the number of distinct elements of the multiset

by one; inserting an element already in the multiset does not; and regardless of whether the element is already in the multiset or not, the multiset's size increases by one.

Other specification languages for sequential programming are VDM (Vienna Development Method) and Z (pronounced "zed"). The latter is based on mathematical set theory and predicate logic, and has been used to specify industrial projects, particularly in Europe. There are also specification languages for concurrent programming (*q.v.*), which are less widely used, but have been applied to some safety-critical problems.

Bibliography

1985. Guttag, J. V., Horning, J. J., and Wing, J. M. "The Larch Family of Specification Languages," *IEEE Software*, **2**, 5 (September), 24–36.

1990. Wing, J. M. "A Specifier's Introduction to Formal Methods," *IEEE Computer* (September), 8–24.

1994. Morgan, C. C. *Programming from Specifications*, 2nd Ed. Upper Saddle River, NJ: Prentice Hall.

Jeannette M. Wing

PROGRAM VERIFICATION

For articles on related subjects *see* AUTOMATIC PROGRAMMING; FORMAL METHODS FOR COMPUTER SYSTEMS; LOGICS OF PROGRAMS; LOOP INVARIANT; MODEL CHECKING; PROGRAM SPECIFICATION; SOFTWARE TESTING; and STRUCTURED PROGRAMMING.

It is important to know that a computer program meets its specifications. Program errors might result in the loss of life or limb, the loss of information, or the loss of financial assets. With the massive penetration of computing technology into society, program errors can result in widespread inconvenience and risk (Neumann (1994) discusses and catalogs numerous failures of computing systems.) Various techniques can be used to determine whether a program satisfies its precise and rigorous specifications. Each technique provides varying amounts of assurance.

The most common technique is *testing* a program (*see* SOFTWARE TESTING). Sample data, presumed to be representative and to cover the necessary extreme cases, is given to the program and the results are compared against known or expected answers. The major problem is to know when to stop testing—how much more assurance of meeting specifications would be gained by additional cases. Or, as Dijkstra (1972) wrote, "Program testing can be used to show the presence of bugs, but never to show their absence!"

In contrast, but often as a supplement to testing rather than as a distinct alternative, is the technique of *program verification*. As that term is used in this article, to *verify* a program means to demonstrate, via a mathematical proof, that the program is consistent with its specifications. It may be quite useful just to prove limited properties, such as that the program terminates (and without executing an operation whose result is undefined, such as division by 0) or that certain variables remain unchanged. The criterion of success requires a sufficiently believable proof, as do all mathematical proofs. Failure to complete the proof may be due to a problem with either the program or the specifications, as well as because of insufficient information about the problem domain or even actual inability to prove a true theorem.

Basic Technique and Example

The most common technique for verifying a program is known as the method of *assertions* (or *invariant assertions* or *inductive assertions*). The basic idea is to associate assertions with various points in the program. *Assertions* are propositions involving the variables of the program usually expressed in a system like the first-order predicate calculus (*see* DISCRETE MATHEMATICS). The intent is that each assertion be a true statement every time the execution of the program passes the point with which that assertion is associated. The proof requirement is to demonstrate that this intent is actually satisfied. Those assertions that appear at the end of a program are often called *postconditions*; assuming that the program terminates, these give the result of the program. Assertions that appear at the start of a program are called *preconditions*. Because programs do not accept arbitrary inputs, a precondition is intended to give a sufficient condition for the program to compute its result. For example, a program to compute the inverse of a matrix or the reciprocal of a number requires nonzero input and perhaps other conditions as well. The only other requirement on the association of assertions is that (the path formed by) every loop must have at least one point with an assertion. An assertion that is true for every execution of a loop is called a *loop invariant*. Such invariants can often be deduced from the program or, indeed, the loop can be constructed to preserve a previously given invariant. In either case, the loop invariant is an essential part of understanding why the program works as well as an essential ingredient of the verification.

The standard way to achieve the proof requirement is to focus on a particular assertion, say P_1, and to follow the program execution from P_1 along all possible paths, stopping on each path when another assertion, P_2, is reached (P_2 is often P_1 again if the path is a loop). One must show, for each such path, that P_1 and the effects of the statements between P_1 and P_2 imply that P_2 holds. Suppose we do this for all assertions,