Modular Verification of Data Abstractions with Shared Realizations

George W. Ernst, Raymond J. Hookway, and William F. Ogden

***Abstract*—This paper presents a method for the modular specification and verification of data abstractions in which multiple abstract objects share a common realization level data structure. Such *shared realizations* are an important implementation technique for data abstractions, because they provide for efficient use of memory; i.e., they allow the amount of memory allocated to the realization of an abstract object to be dynamic, so that only the amount of memory needed for its realization is allocated to it at any one time. To be explicit, an example of this kind of data abstraction is given. Although a number of programming languages provide good support for shared realizations, there has been limited research on its specification and verification.**

An important property of our method is that it allows data abstractions to be dealt with modularly; i.e., each data abstraction can be specified and verified individually. Its abstract specification is made available for use by other program modules, but all of its implementation details are hidden, which simplifies the verification of code that uses the abstraction. We have developed semantics for data abstractions and our method of specification, and have used it to prove that our verification method is logically sound and relatively complete in the sense of Cook [7].

The use of shared realizations impacts specification and verification in several related ways. The manipulation of one abstract object may inadvertently produce a side effect on other abstract objects. Without shared realizations, such unwanted side effects can be prevented by scoping rules, but this is not possible with shared realizations. Instead, the absence of such side effects must be explicitly proven by the verification method. This requires the specification language to provide for quantification over the currently active (allocated) instances of an abstract type that is not necessary for the specification of less advanced implementations of data abstractions.

***Index Terms*—Data abstraction, program correctness, formal methods, specification, verification, and program semantics**

I. Introduction

THIS PAPER describes a modular method for verifying data abstractions in which multiple abstract objects share a common realization level data structure. Such shared realizations of data abstractions arise frequently in space-efficient software, and any effective verification system must have a sound and an efficient way to deal with them. For example, in an application that uses several stacks, some of the stacks may at times contain large numbers of entries; but it often happens that at those times, the other stacks contain only a few entries. The net effect is that at all times, the current aggregate size of all of the stacks is far smaller than the sum of their maximal sizes. So, allocating the maximal amount of memory to each stack would be very wasteful. To make matters worse, often the maximal size of a stack is not known when the stacks are declared; i.e., it is a run-time quantity. The normal solution to this problem is to store all stacked items in a common memory pool, which is usually heap memory, so that each stack is allocated only the amount of memory that it currently needs. This kind of implementation of a data abstraction is a typical example of a *shared realization*, because portions of realizations of more than one abstract object are stored in a single realization data structure.

A number of programming languages provide good facilities for implementing shared realizations, e.g., Modula-2 [48] and Ada. There is very little discussion of specification and verification of shared realizations in the literature, however, even though they raise some interesting issues not found in less advanced realizations of abstract types. For example, a procedure that manipulates an abstract object now has the potential to inadvertently modify another abstract object that is not one of its input parameters. The source of this new problem is that portions of the realizations of both objects are stored in a common data structure, all of which can be accessed by the procedure. Although this is an undesirable complication, it is the price to be paid for the memory conservation advantages of shared realizations. In less advanced realizations, such unwanted "side effects" are prevented by scoping rules, but this is not possible with shared realizations. Consequently, the absence of these side effects must be established by the verification system.

One important feature of our verification method is that it is *modular* in the sense that a realization for a data abstraction can be verified independently of the other parts of a software system that use it. These parts have access to the abstract specification of the data abstraction, but not to its implementation details. This is very important because the main goal of data abstraction is to achieve conceptual simplification by hiding the implementation details from the clients of abstractions. It is particularly important that the details of shared realizations be hidden, because they are normally more intricate than less advanced realizations, and hence are more difficult to verify as well. Fortunately, these details have to be established only once when verifying that the implementation of the abstraction is correct. The correctness of each usage of the abstraction then depends only on its abstract specification, not at all on its implementation details.

Manuscript received July 6, 1992; revised October 1993. This work was supported in part by the National Security Agency under Contract MDA904–92–C–5138. Recommended by R. Kemmerer.

G. W. Ernst is with the Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH 44106.

R. J. Hookway is with Digital Equipment Corp., Acton, MA 01720.

W. F. Ogden is with the Department of Computer and Information Science, Ohio State University, Columbus, OH 43210.

IEEE Log Number 9216607.

0098–5589/94$04.00 © 1994 IEEE

Our verification method is described in the context of Modula-2, but it is applicable to other languages such as Ada that provide for shared realizations. Modula-2 also offers the advantage that its abstraction mechanism is reasonably simple, but still quite powerful. Because the focus of this paper is data abstraction, it does not deal with concurrency, because it would considerably complicate the issues. For expositional purposes, we present an example of a shared realization and describe how it can be verified by our method. Although this example is in Modula-2, the basic features of the example and its verification apply to other languages that support shared realizations.

The next section describes related research. Section III gives an informal overview of our approach to the specification and verification of shared realizations. It describes the main issues presented by shared realizations and outlines our approach for dealing with them. Section IV describes our method for specifying data abstractions that may have shared realizations, and an example of this kind of specification is given in Section V. Our method for verifying such data abstractions is described in Section VI, and Section VII describes its application to the example in Section V, which uses shared realizations. We have proven that our method is logically sound and relatively complete[1] in the sense of Cook [7]. Section VIII gives an overview of the development of these results and the associated metatheory. The last section contains some concluding remarks.

II. Related Work

Modular verification allows one module of a program to be verified independently of the other modules. Because one module can reference other modules, specifications of the latter are needed for verification of the former. The implementation of a module is available only for its verification, however, whereas its specification is available for verification of any module that references it. Many researchers, including the authors, feel that any practical verification method will be modular (see, e.g., [25] and [50]). Structured programming has essentially the same modularity requirement, and the main purpose of data abstraction is to make the external specifications of modules independent of their implementation details. Thus, any nonmodular approach to the verification of data abstractions defeats, at least partially, the purpose of the abstractions in the first place.

There are two basic approaches to the specification of data abstractions. *Algebraic specifications* define abstract operations by means of algebraic axioms, which are equations whose terms contain operations applied to arguments. The other basic approach, sometimes called *abstract models*, uses abstract input–output (I–O) assertions of procedures that manipulate abstract objects to specify data abstractions. The specifications of these procedures are abstract in the sense that their view of the objects being manipulated is an abstraction of the data that is actually stored in the machine. Part of the specification is a function that maps this actual data into its abstract representation.

There are two basic mechanisms that programming languages use to support data abstraction: First, the *class construct* associates a set of procedure and variable declarations with an abstract type, and each of its instances gets a copy of these variables, whose values are the realization of the abstract type instance. The associated procedures manipulate this instance by manipulating the variables used in its realization. This kind of abstraction has its origin in Simula [8] and was used by Hoare [27] when he introduced the abstract model approach to data abstraction. Alphard [49] and Mesa [18] also use this mechanism for data abstraction. Second, *exporting types* from modules can be used to achieve data abstraction by considering an exported type to be an abstract data type. When an abstract type instance is declared (outside the module), an instance of the exported type is allocated and used in the realization of the abstract type instance. Procedures exported from the module that have parameters of the exported type can be used to manipulate this abstract type instance. This mechanism is used for data abstraction in Modula-2, Ada, and Euclid [31]; but Ada and Euclid also can achieve data abstraction by a variant of the class construct. In some cases, the two mechanisms can be used together quite effectively. A nice description of these two methods for abstraction is given in [5].

Exporting types has several advantages over the class construct. Variables local to a module can be used, in addition to an instance of an exported type, in the realization of the instance, because exported procedures have access to such variables. Because such variables may also be used in the realization of other abstract objects, this mechanism provides for the shared realizations described in Section I. The class construct normally does not provide for such shared realizations. Instances of different abstract types may also share memory in the same way, because several abstract types may be defined in a module by exporting several types. A procedure may manipulate several instances of the same or different abstract types, because the type of any of its parameters may be an exported type. In addition, an abstract type may be associated with the name of the module and exported from it to define an abstract object whose realization is stored in variables local to the module. Such an abstract object is the only instance of the exported type unlike other exported types, which may have multiple instances. This provides an implementor with the ability to restrict the number of instances of an abstract type to a single instance.

Allowing several abstract objects to have parts of their realizations in a single data structure is a feature that has not been adequately addressed by research on the specification and verification of data abstraction. This feature permits a procedure that manipulates an abstract object to produce unwanted side effects on other abstract objects, because the procedure can modify the shared data structure. Verification must prove that such unwanted side effects do not occur, but this essentially requires some kind of quantification over abstract type instances that are currently active or allocated;

[1] Most verification systems cannot be logically complete, because they have the integers embedded in them. Intuitively, Cook's [7] concept of relative completeness means that all of the incompleteness is due to the axioms for integers, not to other aspects of the verification system.

i.e., it must be proven that *for all* other instances of the abstract type, its input value to the procedure is the same as its output value. Most specification languages do not provide for quantification over all instances of a type. Verification rules have been developed for Euclid [33], but they deal only with abstractions defined by the class construct. Alphard has a provision for the realization of abstract type instances to share memory, but the verification rules for Alphard deal only with the case when the realizations do not share memory [49]. Similarly, the Gypsy verification rules do not address this feature [30]. The Anna specification language [35] provides for data abstraction by exporting types, and has some facility for quantification over instances of an abstract type. It is not clear, however, how Anna addresses issues surrounding shared realizations, such as preventing unwanted side effects. The associated approach to verification [34] is based on the use of history sequences, and thus is very different from our approach. Ernst and Ogden [12] gave a method for specifying shared realizations, but the language was not powerful enough for the verification that procedures did not produce unwanted side effects. Spitzen and Wegbreit [44] discuss related issues in the context of implementing abstract data types in EL0.

Data abstractions based on the class construct do not have the shared-memory feature, and thus their verification is much better understood. Hoare's [27] original method solved the basic problem, and others have adapted his methods to their language; e.g., the verification method of Alphard [49]. In the method developed in [16], modules are verified by proving statements about their semantics. Thus, this method applies to abstractions based on either the class construct or exporting types, because it is independent of a particular programming language. It does not, however, consider the possibility of the kind of side effects described above.

Data abstraction is supported by object-oriented programming languages, e.g., Smalltalk [19], Eiffel [37], and C++ [46]. The kinds of data abstraction described above are used in object-oriented programming, and in that sense our research results are directly applicable to object-oriented programming. C++ even has a special features that supports shared data in classes. However, object-oriented programming also employs another important mechanism, *inheritance*, which is not part of the conventional programming languages discussed above. Very little work has been done on the verification of programs that use inheritance, and it is beyond the scope of the research described in this paper. Our work is relevant to the specification and verification of object-oriented programs, however, because both use the same kind of data abstraction.

Research on what is sometimes called *data refinement* addresses many of the same issues as data abstraction, but the approach is quite different (see, e.g., [2], [3], [22], [38]). This research attempts to transform abstract specifications into something closer to an implementation. Part of the input to a transformation is the relationship between abstract objects and the data structures that are used to realize them. This is an interesting approach because implementations that are derived in this way do not have to be verified, since the transformations are designed to preserve correctness; i.e., if the input to a transformation is correct, then so its output. The relationship of data abstraction research to data refinement is that the conditions required by verification rules must in some way be enforced by the transformations so that they preserve correctness. Some research on rapid prototyping also uses this approach, but the efficiency of the generated implementations is not one of the primary objectives (see, e.g., [29]).

Research on algebraic specifications (e.g., [4], [10], [21], [11]) seems to be concerned with very different issues than research on the abstract model approach to data abstraction. A major source of these differences seems to be that algebraic specifications do not attempt to deal directly with the implementation of a data abstraction, but focus on the specification of abstract data types and their associated operations. This distinction is made very explicitly in the Larch project [23] (see also [32]). It is based on a two-tiered approach to specification: the Larch Shared Language and a Larch interface language. Specifications in the shared language are essentially the usual kind of algebraic specifications. A Larch interface language is used to specify the procedures that implement abstract operations defined in the Shared Language specifications. The interface specifications are essentially the usual kind of abstract I–O assertions, e.g., like those in [27]. These procedures are then implemented in a programming language like Pascal, and the realization of an abstract object is stored in ordinary program variables. Part of the interface specifications is a function that maps the values stored in these program variables to the abstract value of the object. The Larch project shows the relationship between the two approaches to data abstraction: Algebraic specifications are used to define the abstract types and their operations. The interface specifications are based on the abstract models approach, because they contain invariants, abstract I–O assertions of the operations being implemented, and a function that maps stored values into abstract values. An alternative approach is to use the algebraic axioms to directly synthesize an implementation. Semiautomatic software tools for this and and the correctness proof are described in [17], [24], and [39].

Most theoretical research on verification does not provide for modularity, as defined at the beginning of this section, and cannot be easily modified to do so. The reason for this is that modularity requires the validity of a program to depend on the specifications of externally defined quantities, such as procedures and data abstractions, but not on their implementations. Most definitions of validity, such as those in [6], [7], [9], [20], and [40], do not provide for such external specifications. Instead, the validity of a program depends only on its I–O assertions. In a valid modular program, the precondition of an externally defined procedure must be true whenever it is invoked. Usually, validity does not use such specifications to determine the effects of a procedure call, but rather uses the body of the procedure for this purpose. Our definition of validity does provide for modularity, but changing such fundamental definitions as validity has a large impact on many aspects of the metatheory. As far as we know, the verification methods for the abstract model approach to data abstraction have not been proven to be relatively complete in the sense of Cook [7]. The modularity and relative completeness of our method are new results, particularly because

they deal with shared realizations. Realitive completeness means that any correct program can be verified if we are given a complete deduction system for integer arithmetic. Consequently, any incompleteness of the system is related to some aspect of arithmetic, instead of any feature of the programming language.

III. Overview of Specifying and Verifying Shared Realizations

Dynamic memory allocation is one of the main reasons to implement data abstractions with shared realizations, and frequently heap memory is used in such implementations. Consider, for example, an abstract data type whose instances are sets of integers. An instance of this type can be realized by a linked-list of the elements in the set. This realization allocates only enough memory to a set to store all of its elements, and typically heap memory is used for this realization. Essentially, the realization of the sets share heap memory, and when elements are deleted from one set, the memory can be reused to realize other sets. Procedures are defined that manipulate the abstract sets; e.g., remove(i, x) removes element i from set x, and union(x, y, z) assigns the value of $x \cup y$ to z.

Since any procedure can modify the link of any element of a list, it may accidentally produce side effects on sets that are not parameters to it. Verification must prove that this cannot happen, because it cannot be prevented by scoping rules. There should also be sufficient specification of the realization of sets, however, so that the verification can detect errors in the implementation. In our example, suppose that *union* was implemented by adding the elements of x that are not in y onto the front of the list y and making this new list the value of z. Sets x, y and z all have their correct value, and thus this implementation appears to be correct. The problem is that removing an element of y now produces a side effect on z, because the y list is the last part of the z list. Although verification must prove that such things cannot happen, the real problem is that the specification of the realization should say whether the lists are disjoint. In fact, there are two correct implementations: one uses disjoint lists, and *remove* modifies the lists. In the other implementation, one list may point into the middle of another list, but *remove* never modifies a list; instead, it creates a new list whose last part may be the same as the last part of the input list. If the lists are disjoint, this should be stated in the specifications, so that the implementation can rely on this property. But, of course, this property must also be maintained by the implementation. Our specification method requires that this disjoint property is part of the invariant of the module in which the abstract sets are implemented.

Although sharing memory is desirable because it provides for efficient use of memory, it complicates both the specification and the verification of a data abstraction. A basic facility that is needed is some means of quantifying over all instances of an abstract type. In the above example, this is needed to specify that the lists are disjoint; i.e., we must specify that *for each list*, none of its list elements are list elements of other lists. This specification, which is part of the module invariant, must be maintained by all procedures exported from the module. In our specification language, the module invariant is the only place where the specification of an abstract data type can quantify over all of its instances. Our verification method, however, uses such quantification for a number of reasons. The most obvious is that the absence of side effects on procedures that manipulate abstract objects is expressed by the following condition: *For all instances* x of an abstract type that are not parameters to a procedure, the input (abstract) value of x must equal the output value of x.

Most specification languages in the literature (e.g., [33]) do not provide for such quantification, and for a good reason: It appears not to be needed when the realizations of abstract objects do not share memory, and using it appears to complicate both specifications and verification. It seems to be necessary, however, when realizations share memory, and the complication is the price that must be paid. Because this kind of implementation is very common in contemporary software, the price seems to be justified.

To achieve this quantification in our specification language, we let R denote an array of the values of all of the instances of an abstract type. In the above example, $R[x]$ is a pointer to the list for set x. The size of R is denoted by *alloc*, which is the number of instances of the abstract type. When a new instance of the type is declared, *alloc* is increased by 1. This permits quantification over the instances by a formula of the form $\forall x(1 \le x \le \mathit{alloc} \rightarrow (\cdots R[x] \cdots))$. Thus, the specification and verification takes place in a larger name space than the implementation, because, essentially, R gives us a name for each instance of the abstract type. The verification rules modify the code to keep the names consistent. For example, if an instance x of the abstract type is a parameter to a procedure, then an x in the procedure body is replaced with $R[x]$, which is the new name of x for the purpose of verification.

The above discussion is a high-level overview of our approach to the specification and verification of abstract types. The actual verification rules, with all of the formal details, are given in Section VI. To be concrete, Section V contains an example of the specification and implementation of a data abstraction that uses shared realizations. The example does not use heap memory, because it would require extensive use of pointers, which would unnecessarily complicate both specification and verification. Although the heap memory implementation described above is more realistic, the example in Section V illustrates most of the important concepts, as well as giving some idea of the variety of implementations that any general method must be capable of handling.

IV. Specification of Data Abstractions

Our approach assumes that each data abstraction has a complete specification as well as an implementation. Such a specification consists of declarative formulae that are inserted into the implementation at appropriate places, which is a common approach found in the literature, e.g., [33]. Thus, a data abstraction is a mixture of code and specifications. The purpose of this section is to describe our method of specification, i.e., the various components of a specification and where they are positioned in the code. Modules that use an externally

```
1.  module M: TM;
2.    export M, AT, q;
3.    import gv;
4.    require MPre;
5.    ensure MPost;

6.    MDec1;
7.    var lv: T1;
8.    MDec2;

9.    abstract type AT;
11.     abstract structure as: T2;
12.     realization structure rs: T3;
13.     correspondence CT;
14.     invariant IT;
15.     initialization (cp : T4);
16.       require IPre;
17.       ensure IPost;
18.       IBody
19.     end initialization;
20.     finalization; FBody end finalization;
21.   end AT;

23.   correspondence CM;
24.   invariant IM;

25.   procedure q (cp : AT; var vp : AT);
26.     require QPre;
27.     ensure QPost;
28.     QBody
29.   end q;

30.   begin MBody
31. end M;
```

Fig. 1. The declaration MD of a typical module that defines and exports an abstract type as well as an abstract object.

defined data abstraction have access to the specification of its abstract view, but not to its implementation or specification about its realization. This section also describes our method for specifying such externally defined data abstractions.

The specification language has first-order logic and number theory embedded in it; i.e., arithmetic symbols receive their standard interpretation. It may also contain a few other built-in theories, e.g., set theory. The specification language has a mechanism for defining new predicates and functions by using explicit definitions that are nonrecursive. Recursive definitions are useful and could be added to the language, but they are not needed for this paper. For readability, we use ordinary mathematical notation in specifications, but this should be considered to be just a readable form of actual statements in the specification language.

Our method for specifying and implementing data abstraction is depicted schematically in Fig. 1; e.g., the name *QBody* denotes the sequence of declarations and statements that constitute the body of procedure q. The module M declared in this figure defines one abstract type AT, whose instances can be declared by program statements that use this module. M also exports one procedure q, which manipulates these instances, and *QPre* and *QPost* are the abstract specifications of q. The importance of Fig. 1 is that it contains one use of each specification or implementation construct for the kind of data abstraction for which our verification method is designed. Thus, Fig. 1 gives a total picture of this kind of data abstraction, but many of the constructs in Fig. 1 can have zero or multiple occurrences in actual use; e.g., usually, there will be several exported procedures for manipulating instances of an abstract type defined in a module. The remainder of this section describes the form and intent of each of the constructs in Fig. 1, and how they interact with one another.

The module name M in Fig. 1 is used to denote an abstract object defined by the module. It is the only instance of the type TM. For example, in a program that used only one stack, M could be used to denote the stack, and TM would be the type of the stack, but M would be its only instance. A module that defines such an abstract object has a type following its name; a module such as the one in Fig. 3 does not define such an abstract object, and thus does not have a type following its name. Normally, TM is not a built-in type, but rather a type that Modula-2 does not provide for directly such as a set of strings. Thus, the first part of line 1 is required by Modula-2, but the ": TM" is part of our specification method.

The export and import lists are Modula-2 statements. The precondition of the module *MPre* specifies a condition that must be satisfied by the value of the global variable gv that is imported by the module. In general, 0 or more global variables can be imported by a module, but they are all typified by the single global variable imported by M. The declaration of M can update the value of gv, and thus the module's postcondition *MPost* is a statement about the new value of gv; $\#gv$ in *MPost* denotes the value of gv before the declaration. *MPost* is also a statement about the abstract object M that specifies properties of its initial value.

MDec1 denotes the usual constant and type declarations of a module, and lv is a local variable that is used in the realization of abstract objects. *MDec2* is the declaration of procedures that are not exported from the module. These procedures are utility routines that are used in other parts of the module, e.g., in the implementation of procedures that are exported from the module. Thus, lines 6–8 are just Modula-2 code.

AT is an abstract type that is exported from the module. Line 11 specifies the abstract view of an instance of AT to be of type $T2$, which normally is not a built-in type of Modula-2. This line also specifies that as is the formal name of an abstract instance of AT in lines 9–21. Similarly, line 12 declares $T3$ to be the realization type of AT, which will be a Modula-2 type. An instance of $T3$ is allocated when an instance of AT is declared. This line also specifies that rs is the formal name of a realization instance of AT in lines 9–21. Line 13 gives the relationship between an abstract instance of AT and its realization data. Since part of the realization can be stored in the local variable lv (the example in the next section illustrates this), CT is a statement in the specification language that refers to as, rs, and lv. We also require that this relationship is a function from the realization space to the abstract space; i.e., the values of rs and lv uniquely determine the value of as. Of course, the inverse relation is not a function, because there may be many different realizations of a single abstract value. Invariant properties of the realization of an instance of AT are specified in line 14; thus, IT is a statement about rs and lv. This invariant will be satisfied when the module is entered or exited, but executing code inside the module may temporarily violate the invariant.

Lines 15–19 give the specification and code for initializing an instance of AT. This initialization is essentially a procedure that is automatically invoked when an instance of AT is declared. The procedure can have an input parameter cp whose actual value is given in the declaration of an instance of AT, and has the instance itself as an implicit parameter. In the body of this procedure *IBody*, rs refers to the instance being

declared. Line 17 gives the postcondition of this procedure that specifies properties of the initial abstract value of the instance. Thus, *IPost* is a statement in the specification language about as (and *cp*, too). Line 16 is the precondition of the procedure that constrains the values permitted for *cp*. Line 20 gives code that is automatically executed when an instance of AT is deallocated. This is essentially a procedure with an instance of AT as an implicit parameter, which is denoted by rs in *FBody*, the body of the procedure. This procedure has no pre- or postconditions, because its execution is transparent to the abstract level. It serves a very important bookkeeping function, however, which is specified by the module invariant. This is illustrated by the example in the next section.

Most of the information in the declaration of AT consists of specifications of one form or another. In Modula-2, without our specifications, the declaration of AT would be as follows:

$$\textbf{type}\ AT = T3,$$

and initialization and finalization would be declared as separate procedures. Everything else is specifications. In Modula-2, the responsibility of invoking the initialization and finalization procedures would fall on the code declaring an instance of AT. This would present a number of difficulties, such as the fact that IT normally would not be true of a new instance of AT. This is the reason why we added their automatic invocations to Modula-2.

Line 23 specifies the relationship between the value of the abstract object M and its realization. An abstract object is *realized* by some or all of the values stored in one or more program variables. The specification of the abstract object indicates which components of which variables are used in the realization and how their values map to the abstract value of the object. In the case of M, it is realized by values stored in program variables local to the module, which are typified by lv in Fig. 1. Thus, CM is a statement about M and lv. As with CT, we require this relationship to be a function from the realization space to the abstract space.

The module invariant in line 24 specifies invariant properties of data used to realize abstract objects. This invariant will be satisfied when the module is entered or exited. An obvious kind of invariant specified by IM is a property of the realization of M in lv. A more subtle, but equally important, kind of property specified by IM concerns details of the realization of different instances of AT. For example, if such instances are represented by linked lists, we may want to specify that no two instances of AT point to the same list. To do this, we need to refer to the realization of more than one instance. This is not possible in IT, which is a statement about only one instance. Statements in IM can use the notation $AT.R[i]$ to refer to the ith instance of AT. The specification language interprets $AT.R$ as an array of the realization instance of AT; each element is of type $T3$, and the subscripts range from 1 to *AT.alloc*. This mechanism allows quantification over the instances of an abstract type, and thus can be used to specify properties such as no two realization instances have the same value. The example in the next section provides the reason why this is needed, and illustrates how it is done in our specification language.

Procedure q has two instances of AT as parameters, the second of which can be modified by q. *QPre* is the precondition of q, which is a statement about the abstract values of both cp and vp. The postcondition *QPost* is also a statement about these parameters, but here vp refers to its output value. The input value of vp is denoted by $\#vp$ in *QPost*. The abstract object M is interpreted as a global variable that may also be manipulated by q, and thus M can occur in both the pre- and postconditions of q. In the latter, it refers to the output value, and $\#M$ refers to the input abstract value of M. *QBody* is the Modula-2 code that implements q, and thus cp and vp in *QBody* look like ordinary program variables of type $T3$.

It is important that the pre- and postcondition of q are statements about the abstract values of its parameters, because they are the specification to the external world that may invoke q. Of course, the code that uses q should have no knowledge of its implementation details. On the other hand, it is important that the invariants are statements about realization data, because some of the properties are transparent at the abstract level. In some cases, it is also quite useful to have invariants that are statements about abstract values. For simplicity, we have not provided for this, because such invariants can be expressed in terms of realization values.

MBody initializes the local module variables and perhaps updates the variable imported by the module. The former is the mechanism by which the abstract object M is initialized, which is specified by *MPost*.

One module can define zero or several different abstract types, and procedures can have arbitrary combinations of these as parameters. A module has the option whether it will define a single-instance abstract object like M. The combination of such possibilities provides considerable variety. The module in Fig. 1, however, typifies the kind abstraction provided by Modula-2 and our method for specifying such abstractions, because it contains one of each kind of programming construct involved in the abstraction; e.g., q contains one constant parameter of type AT and one variable parameter. The generalization to 0 or multiple uses of such constructs is straightforward. Normally, there will be multiple procedures like q for manipulating abstract objects, but their implementations and specifications will be done in the same way as q's.

An important aspect of our approach is the modularity of the specification and implementation of data abstractions. An externally defined module can be used by any other module by including the *external* specification of the former in the latter. This specification is just a module declaration, with all of the implementation related information removed. For example, Fig. 2 gives the external specification MS of the module declaration in Fig. 1.

For simplicity, the entire text in Fig. 2 is inserted into a client module, but a mechanism like the *include file* of the programming language C would be more practical, because there may be multiple clients. The external specification of a module contains no declarations local to the module that are not exported, or any program statements. In addition, all specification about invariants and correspondences are omitted from the external specification, because they depend

```
module M: TM;
  export M, AT, q;
  import gv;
  require MPre;
  ensure MPost;

  abstract type AT;
    abstract structure as: T2;
    initialization (cp: T4);
      require IPre;
      ensure IPost;
    end initialization;
  end AT;

  procedure q(cp: AT; var vp: AT);
    require QPre;
    ensure QPost;
  end q;

end M;
```

Fig. 2. The external specification MS of a typical module which defines and exports an abstract type as well as an abstract object. This is the external specification (without implementation) of the module declaration in Fig. 1.

on realization values that are hidden from the abstract level. The finalization of abstract types is absent, because it is transparent at the abstract level. We use Fig. 2 to schematically describe our method for specifying the external view of a module specification in the same sense that Fig. 1 describes our method for specifying Modula-2 data abstractions, as discussed above.

V. An Abstract Data Type Example

This section contains an example of a module that defines an abstract data type whose instances have shared realizations. This illustrates what we mean by shared realizations, and why they are important. It also illustrates our specification method. In Section VI, our verification rules are applied to this implementation and specification to illustrate the use of our method. This example nicely illustrates the definition of an abstract data type that can have multiple instances, but the module name does not represent an abstract object, and thus this aspect of our method is not illustrated by Fig. 3. The reason for not illustrating both of these features is that such examples are too involved for the purpose of exposition. In fact, even the example in Fig. 3 is more involved than desired, but we could not find a simpler example that illustrates the concepts being addressed.

Fig. 3 contains a module declaration that defines *IntegerSet*, which is an abstract data type whose instances are sets of integers. For readability, we have omitted some of the straightforward Modula-2 code, which is indicated by ellipses. The abstract view of integer sets in this example is essentially the same as the one in [27], but the realization of the latter is very different, because each instance is allocated a separate block of memory. For the purpose of exposition, this example does not make use of heap memory in order to avoid extensive use of pointers. This also removes issues about the kind of garbage collection supported by the programming language. Although this is a considerable simplification, the example in this section illustrates most of the important issues underlying shared realizations. A more realistic implementation would use heap memory, however, and Section III outlines how our method would deal with such implementations.

In our realization, the elements of the various sets are stored in the *mem* array. Each set is assigned a unique name, and if $mem[i].id$ is one of these names, then $mem[i].val$ is an element of that set. $mem[i].id = 0$ indicates that $mem[i]$ is not currently being used for storing an element of a set, and all other elements of *mem* are used for storing elements of sets. The integers from 1 to *IdSize* are used as the names of sets. *FreeId* indicates which of these set names is currently in use; i.e., if $FreeId[i]$ is *true*, then i is not currently being used as the name of a set.

The primary advantage of this implementation is that a fixed amount of memory is not allocated for each set; rather, each set uses as many elements of the *mem* array as necessary. Thus, small sets will use a small portion of *mem*, and large sets will use a much larger portion of *mem*. In this sense, the sets share the storage capacity of the *mem* array, which may be much smaller than the sum of the maximal sizes of all of the sets. This is accomplished by reusing the memory for elements that are deleted from sets to store elements that are subsequently added to sets. The verification of this implementation requires that operating on one set does not accidentally change the value of another set. This is possible because a procedure that can change one element of the *mem* array can change any other element, too, and thus may produce side effects on other sets. The absence of such unwanted side effects must be explicitly verified, because they cannot be prevented by scoping rules. These features of the implementation are explicit in the code in Fig. 3, whose description follows.

Line 2 of Fig. 3 specifies that the abstract data type *IntegerSet* is exported from the module, along with some procedures that manipulate its instances. Lines 3–10 declare constants, types, and variables that are used in realizing *IntegerSets*, as described above. The i and j declared in lines 11–12 are used by the code in the body of the module. Three procedures are declared in lines 13–27 that are not exported from the module, because they do not manipulate *IntegerSets*; rather, they are used in the implementation of the exported procedures. To avoid forward references, the declaration of these procedures comes before the declaration of *IntegerSet*, because it contains some code that references these local procedures, e.g., *erase* on line 42.

Line 29 declares the abstract value of an *IntegerSet* to be a set of integers, and line 30 declares its realization value to be of type *IdRange*, an instance of which is allocated whenever a new instance of *IntegerSet* is declared. Lines 29–30 also define as to be the formal name of an abstract instance of *IntegerSet*, and rs to be the formal name of the corresponding realization instance. These names are used in the remainder of the declaration of *IntegerSet*. Line 31 specifies the relationship between rs and as: Each component of *mem* whose id field is rs contains in its *val* field an element of the abstract set as. *Rep* is a function defined for the specification of this module; its definition is given in Fig. 4. The specification language provides for explicit definitions like those in Fig. 4.

Line 32 specifies properties that each realization instance rs possesses. The second conjunct requires rs to be a name that is currently in use according to *FreeId*. $Distinct(rs)$ specifies that each element of the integer set whose realization is rs is

```
1. module IntegerSetModule; (*The purpose is to define abstract type IntegerSet*)
2.    export IntegerSet, insert, remove, has, union, intersect;

3.    const IdSize=100;
4.       MemSize=1000;
5.    type IdRange=1..IdSize;
6.       MemRange=1..MemSize;
7.       element=record id: 0..IdSize;
8.          val: integer end;
9.    var mem: array [MemRange] of element;
10.      FreeId: array [IdRange] of boolean;
11.      i: MemRange;
12.      j: IdRange;

13.   procedure erase(x: IdRange);
14.      (* Puts 0 in the id field of each element of mem
15.         that has an id of x. *)
16.      ...
17.      end erase;

18.   procedure add(i: integer; x: IdRange);
19.      (* Find an element of mem whose id is 0; make its
20.         id x and its val i. *)
21.      ...
22.      end add;

23.   procedure contains(x: IdRange; i: integer): boolean;
24.      (* Tests if there is an element of mem whose id is x
25.         and whose val is i. *)
26.      ...
27.      end contains;

28.   abstract type IntegerSet;
29.      abstract structure as: set of integer;
30.      realization structure rs: IdRange;
31.      correspondence as=rep(rs);
32.      invariant distinct(rs)&¬FreeId[rs];
33.      initialization;
34.         ensure as=∅;
35.         var i: IdRange;
36.         begin i:=1;
37.            while not FreeId[i] do i:=i+1 end;
38.            rs:=i;
39.            FreeId[i]:=false
40.         end initialization;
41.      finalization;
42.         begin erase(rs);
43.         FreeId[rs]:=true
44.         end finalization;
45.      end IntegerSet;

46.   invariant
47.      ∀i(1≤i≤MemSize&mem[i].id≠0 → ¬FreeId[mem[i].id])
48.      & ∀i(1≤i≤IdSize&¬FreeId[i]
49.            → ∃x(1≤x≤IntegerSet.alloc&IntegerSet.R[x]=i))
50.      & ∀xy(1≤x,y≤IntegerSet.alloc&x≠y
51.            → IntegerSet.R[x]≠IntegerSet.R[y]);

52.   procedure insert(i: integer; var x: IntegerSet);
53.      ensure x=#x∪{i};
54.      begin
55.         if not contains(x,i) then add(i,x)
56.      end insert;

57.   procedure remove(i: integer; var x: IntegerSet);
58.      ensure x=#x−{i};
59.      (* Put a 0 in the id field of the element of mem
60.         whose id is x and whose val is i. *)
61.      ...
62.      end remove;

63.   procedure has(x: IntegerSet; i: integer): boolean;
64.      ensure has=(i∈x);
65.      begin return contains(x,i) end has;

66.   procedure union(x,y: IntegerSet; var z: IntegerSet);
67.      ensure z=x∪y;
68.      ...
69.      end union;

70.   procedure intersect(x,y: IntegerSet; var z: IntegerSet);
71.      ensure z=x∩y;
72.      ...
73.      end intersect;

74.   begin for i:=1 to MemSize do mem[i].id:=0 end;
75.      for j:=1 to IdSize do FreeId[j]:=true end

76.   end IntegerSetModule;
```

Fig. 3. A module that defines the abstract type *IntegerSet*.

$$rep(x) = \{v \mid \exists i(1 \le i \le MemSize \& mem[i].id = x \& mem[i].val = v)\}$$

$$distinct(x) \equiv \forall ij(1 \le i, j \le MemSize \& mem[i].id = x \& mem[j].id = x \& i \ne j \rightarrow mem[i].val \ne mem[j].val)$$

Fig. 4. The definition of predicates and functions used in the specification of the abstract data type *IntegerSet* in Fig. 3.

stored only once in *mem*; i.e., any two components of *mem* whose ids are rs must have different values stored in their *val* fields. *Distinct* is a predicate used in the specification of the module, and its definition is also given in Fig. 4.

Lines 33–40 pertain to the initialization of a new instance of *IntegerSet*. Line 34 specifies that the initial value is $\emptyset$, because as denotes the abstract value of the instance being initialized. Lines 36–39 are the program statements that do the initialization. They find a *true* element of *FreeId* and make its index the value of rs, the realization instance of the type being initialized. This element of *FreeId* is also changed to *false*.

The code in lines 42–44 is executed whenever an instance of *IntegerSet* is deallocated; rs is assumed to be its realization value. This code makes all of the elements of *mem* whose ids are rs unused by putting 0 in their id fields. It also frees the name rs by setting this element of *FreeId* to *true*. This concludes the declaration of the abstract type.

Lines 46–51 give properties that are always satisfied by the local module variables that are used to represent *IntegerSets*. Line 47 requires the name of any set that is stored in *mem* to be in use according to *FreeId*. Lines 48–49 requires any element of *FreeId* that is *false* to be the name of some *IntegerSet*. This requires quantification over all instances of *IntegerSet*. Our specification mechanism for this is to use *IntegerSet.R* to denote an array of the realization values of all of the instances of *IntegerSet*. Its indices range from 1 to *IntegerSet.alloc*. Thus, in line 49, *IntegerSet.R*$[x]$ denotes the realization value of the xth instance of *IntegerSet*, and x is existentially quantified over all such instances. Lines 50–51 require the name of a set to be unique. This also involves quantification over all of the realization values of *IntegerSets*.

Lines 52–56 are the declaration of the procedure *insert*, which is exported from the module. Because it has no explicit precondition, it is *true* by default in our specification language. Line 53 specifies the postcondition of *insert* to be that the output value of x is its input value with the element i added to the set. In an ensure clause, a parameter x denotes its output value, whereas $\#x$ denotes its input value. In both cases, these are abstract as opposed to realization values. On the other hand, an x in the code in line 55, which is the implementation of *insert*, denotes a realization value. This code checks whether i is already a member of the set; if not, i is added to the set. This is done by calling some local (not exported) module procedures, which also consider x to be a realization value. Of course, these procedures reference the *mem* array in which the values of x are stored.

Lines 57–73 declare the other four procedures that are exported from the module. Their declarations are similar to that of *insert*. *Union* and *intersect* have three different *IntegerSets* as their parameters. Lines 74–75 are executed when the module is declared. This code initializes *mem* and *FreeId*; i and j are just temporary variables used for this purpose. The initial value of *mem* has 0 in all of its id fields, and initially *FreeId* has *true* for all of its elements.

An *IntegerSet* is declared like any other variable; e.g., **var** *s: IntegerSet*. It is also passed to a procedure like any other variable; e.g., *insert(3, s)*. The important feature is that the calling environment knows only about the abstract value of s. This is accomplished by allowing the pre- and postconditions of a procedure to refer to only the abstract values of such variables. This makes the invocation of procedures, like *insert*, look identical to that of ordinary procedures.

The realization of *IntegerSets* used in this example is different than standard ones, e.g., the one in [27] and the use of linked lists. An important feature of this example is that local module variables (i.e., *mem* and *FreeId*) are shared by the realizations of the different *IntegerSets*, which provides for efficient use of memory. Although the use of linked lists would have this same feature, their use presents complications that are avoided by this example, such as issues about garbage collection. Also, the code would explicitly manipulate the links of list elements. Thus, incorrect code may create circular lists, which prevents the *rep* function (defined in Fig. 4) from being recursively defined on the links of the list elements. The example in this section, though less realistic than a linked list implementation, illustrates most of the issues involved in shared representations. It also illustrates the variety of implementations that can be used, because it is somewhat nonstandard, and any general method must be capable of dealing with all of them.

VI. Verification of Data Abstractions

This section describes our verification method for data abstractions. This description is based on the kind of specification and implementation in Fig. 1, because it typifies the kind of data abstraction for which our method is designed and the kind of specification required by our method. Although Fig. 1 is based on Modula-2, our method can be easily adapted to other programming languages that support shared realizations, such as Ada.

Our verification method is designed to establish *partial correctness*, which requires that all computed values satisfy their specifications; the method does not establish that computations terminate.

A declaration or program statement is correct in some contexts but not in others, because it may contain external references such as a call on a procedure declared externally. We denote this by a verification formula of the form $C\backslash P$, where P is the program fragment to be verified, and C describes the context in which P executes. P is either a declaration like the one in Fig. 1, which contains its own specification or a formula of the form[2] **assume** $Q1$; B; **confirm** $Q2$. In the latter, B is a sequence of program statements (and declarations) with precondition $Q1$ and postcondition $Q2$. The *context* C is just a list of the specifications of externally defined procedures.

Our verification method is based on verification rules of the form introduced by [26], but our rules are concerned with programming constructs that are used in data abstraction. The rules are designed for doing "backward proof by subgoaling";

[2]This is written as $Q1\{B\}Q2$ in Hoare's [26] notation.

$$\frac{\begin{array}{l} C\backslash MDec2; \\ H1,\ H2,\ H3,\ H4 \end{array}}{C\backslash MD}$$

Fig. 5. The verification rule for modules. MD is the module declaration in Fig. 1. The hypotheses, $H1$–$H4$, of the rule are given in Figs. 8 to 11.

$$\frac{\begin{array}{l} C\backslash MD \\ P \rightarrow MPre \\ C,\ Sq,\ SATI\backslash \textbf{assume}\ \exists x(P[x/gv]\&MPost[x/\#gv]);\ B \end{array}}{C\backslash \textbf{assume}\ P;\ MD;\ B}$$

Fig. 6. The verification rule for declaring a module. MD is the module declaration in Fig. 1 and Sq is the specification of procedure q and *SATI* is the specification of the initialization procedure for AT. x is a variable which does not occur in either P or *MPost*.

i.e., the conclusion (bottom line) of each rule is matched to the code/specifications to be verified, and when they match the hypotheses become subgoals to be proven. The rules are designed so that any subgoal that contains programming language constructs will match the conclusion line of precisely one rule. This makes the subgoaling process deterministic, and hence it is a decision procedure for the programming language part of the system. The terminal subgoals are logic formulae which contain no programming language constructs; these so called *verification conditions* are proven by conventional logical deduction. This method of generating verification conditions is based on [28]. In this section we describe the verification rules for data abstraction; rules for other programming constructs are given in [13].

Our verification rules are designed to process abstract syntax trees of programs, because this preprocessing of characters in the raw code frees the rules from syntactic details and makes the conceptual part of the rule clearer. The rules assume that identifiers have unique names in the abstract syntax trees as the result of resolving their overloading. Therefore, when a new procedure p is declared, its name will be distinct from the names of all of the global procedures whose specifications are listed in the context C. Although the rules process abstract syntax trees, they are described in terms of program text, because the text is more readable than the syntax trees that the text represents.

The rule for verifying the module in Fig. 1 is given in Fig. 5. The conclusion of this rule specifies that the module declaration MD in Fig. 1 is correct when declared in context C. (Recall that the backslash separates the programming statements from the context.) The rule has five hypotheses, each of which is a verification formula that requires some part of the module declaration to be correct: The first hypothesis requires the local procedures that are not exported from the module to be correct. $H1$ requires the declaration of q to be correct; $H2$ and $H3$ require the initialization and finalization of instances of AT to be correct; and $H4$ requires *MBody*, whose purpose is to initialize the local variables of M, which in turn initializes the abstract object M, to be correct. Hypotheses $H1$–$H4$ are the major part of verifying modules, and most of this section is devoted to their description. Before describing them, however, we give the rules for declaring modules and instances of abstract types, which shows the general approach of our method. The verification rule for declaring a module is given in Fig. 6.

The conclusion of this rule requires a sequence of declarations and program statements to be correct. The precondition of this sequence is P, and the first element is the module declaration MD. B is the remainder of the sequence, followed by its postcondition. C specifies the context in which these statements execute. The first hypothesis requires the declaration of MD to be correct in context C. The second hypothesis requires the precondition of the module to be true whenever the precondition of the bottom line of the rule is true.

The final hypothesis requires B to be correct after the declaration MD has been made. This declaration has two effects; the first is that the specifications of the procedures exported by the module are added to the context. This defines their effects when they are invoked by statements in B. The specification Sq of procedure q is just its declaration in MD, with *QBody* removed. The initialization part of AT is essentially a procedure that is implicitly exported from the module because AT is exported from the module. *SATI* is the specification of this procedure. It is the only part of the declaration of AT that is put in the context; other parts, such as the finalization, are transparent to the abstract view of AT.

The other effect of declaring MD is to update global variables that the module imports and to initialize the abstract object M, which is exported from the module. The precondition of the third hypothesis in Fig. 6 specifies that this is done correctly. This precondition requires the existence of an initial value for gv, which satisfies both P and *MPost*. This is specified by substituting a new variable x for gv in P, and for $\#gv$ in *MPost*, and existentially quantifying x; i.e., x does not occur in P, B, or *MPost*. Our notation for substitution is that $[t_1/x_1, t_2/x_2, \cdots, t_n/x_n]$ denotes the substitution of term t_i for variable x_i for $1 \leq i \leq n$. This is precisely defined as *simultaneous proper substitution* in [45]. The precondition of the third hypothesis specifies the properties of gv after the declaration MD, because it contains *MPost*, which is a statement about gv that denotes its value after the execution of MD. This precondition also specifies properties of the initial value of the abstract object M, because *MPost* is also a statement about M; the execution of B assumes these properties of M and gv to be true.

An important feature of our notation for verification rules is that each line (either the conclusion or one of the hypotheses) of a rule is an independent formula, and the proofs of the hypotheses are independent of one another. The scope of each free variable is the line in which it occurs. For example, the gv in the second hypothesis of the rule in Fig. 6 denotes its input value, and it is important that this usage is consistent in both P and *MPre*. The gv in the third line, however, denotes the value of gv after the declaration MD, which is consistent with occurrences of gv in B. The initial value of gv is denoted by x in this line, which can have references to both values. (Note that gv in P and $\#gv$ in *MPost* both refer to the initial value of gv, whereas gv in *MPost* refers to the value after the declaration MD.) The existential quantifier is the formal means by which a fresh or new symbol to denote the input value is introduced.

The verification rule in Fig. 7 is used to verify the declaration of an instance of the abstract type AT defined in the module in Fig. 1. The conclusion of this rule is the declaration of an instance of AT followed by B, which is a sequence of declarations and program statements followed by the postcondition. The first hypothesis requires the precondition of the bottom line of the rule to imply *IPre*, the precondition of the initialization part of AT. *IPre* is contained in the specification *SATI* of the initialization of AT, which is the initialization part of AT in Fig. 1 with *IBody* removed. This is the reason why *SATI* needs to be in the context of the conclusion of this verification rule. The purpose of *IPre* is to specify properties of the parameter to the initialization, which is why the actual parameter *acp* is substituted for the formal parameter *cp* in the first hypothesis of the rule. The second hypothesis requires B to be correct when the precondition of the conclusion of the rule is satisfied and when the new instance x of AT satisfies the initialization postcondition *IPost*, which specifies the initial value of an AT instance. Of course, actual parameters must be substituted for formals in *IPost*.

$$\frac{\begin{array}{l} P \rightarrow IPre[acp/cp] \\ C, SATI \setminus \textbf{assume } P \ \& \ IPost[acp/cp, x/as]; B \end{array}}{C, SATI \setminus \textbf{assume } P; \textbf{ var } x: AT(acp); B}$$

Fig. 7. Theverification rule for the declaration of an instance of abstract type AT which is defined in the module in Fig. 1. *SATI* is the specification of the initialization procedure for AT.

The verification rules in Fig. 5–7 show our approach to verifying modules and programs that use the data abstractions that they define. Implicit in these rules is the fact that programs that invoke procedures that manipulate abstract objects and type instances are verified in the same way as programs that invoke ordinary procedures: The specification of both is in the context, and the same verification rule is used for both. The only difference is that the specification of the former refers to objects whose types are not defined by the programming language, but rather by data abstraction. Next we describe the details of the verification of modules that define data abstractions. These details are quite intricate, primarily because the formulae contain a relatively large number of substitutions that make them difficult to read. The verification conditions for a particular module's implementation, however, are much more readable and are about what one would expect, because the substitutions are removed in the process of generating them. This is illustrated in the next section, which describes the verification conditions for the module in Fig. 3, together with how they are produced.

A. Names Occurring in Verification Formulae

Verification of the module declaration in Fig. 1 places requirements on the various abstract objects and their realizations defined by the module. A method for referring to each of these objects and the various components of their realization is described in this section. This method is used in the formulae for verifying a module declaration, and they contain considerable references to these objects.

R denotes an array of the realization values of the instances of abstract type AT. The bounds on this array are 1 to *alloc*, with the convention that the array is empty when *alloc* equals

0, which is the case immediately after the module declaration. $R[i]$ denotes the realization value of the ith instance of AT The type of $R[i]$ is $T3$. An important feature of this notation is that it allows quantification over all instances of AT by having i range from 1 to *alloc*. An example of this notation is in the module invariant in Fig. 3, which was described in the last section.

The module invariant is the only place in a module declaration where there is quantification over the instances of an abstract type. All other references to either the abstract value or the realization value of an instance of an abstract type is to a given instance, e.g., rs, and the *R/alloc* notation is not needed. In fact, the module invariant is the only place in the module declaration where this notation can be used. This simplifies the declarations of modules that on the one hand makes the programmer's task easier, and on the other hand eliminates this complexity from both the code and the specifications. This notation, however, as well as that described below, is heavily used in the formulae for verifying the modules, which adds to their complexity and that of their proofs. Thus, the name space of the verification formulae and their proofs is larger than that of module declarations.

The verification formulae for procedures that manipulate abstract objects need to quantify over the instances of abstract types, for reasons that are explained in the next subsection. The *R/alloc* notation described above is used for this purpose. In addition, it is extended to include the abstract values of instances. The name A denotes an array of the abstract values of the instances of AT. $A[i]$ denotes the abstract value of the ith instance, and the bounds on A are 1 to *alloc*. In the verification formulae for procedures, the name A refers to the abstract values before the procedure is executed, and the name Ar denotes the array of the abstract values after its execution. This allows quantification over the abstract values both before and after the procedure call.

In general, there may be multiple abstract types defined in a module, and thus we add the type name to these various names described above; e.g., R should really be $AT.R$. The latter notation is used in Sections III and IV; but in this section and the remaining sections, the type name is dropped for readability, with the understanding that it is needed, at least when multiple abstract types are defined in a single module. The use of R, A, Ar, and *alloc* can occur only in specifications, but syntactically they should be treated like reserved words. This allows type checking to be applied to the specifications, in addition to the programming language, which is important in the construction of software tools that process specifications.

As pointed out in the description of Fig. 1, M denotes the abstract value of the single-instance abstract object defined by the module. Because the procedure q may change the value of this object, we use the name Mr for the result of this modification in the verification rules. However, in the postcondition of procedure q, which may manipulate M, $\#M$ denotes the abstract input value of M, and M denotes its abstract output value. This notation is used in the specification of q, and the M/Mr notation is used in the verification formulae for the module.

```
C, SLD \

  assume IM&CM
    & 1≤cp, vp≤alloc&cp≠vp
    &QPre[A[cp]/cp, A[vp]/vp]
    & ∀i(1≤i≤alloc → CT[A[i]/as, R[i]/rs]&IT[R[i]/rs]);

  QBody[R[cp]/cp, R[vp]/vp];

  confirm IM& ∀i(1≤i≤alloc → IT[R[i]/rs])
    & (∀i(1≤i≤alloc → CT[Ar[i]/as, R[i]/rs])&CM[Mr/M]
      → QPost[A[cp]/cp, Ar[vp]/vp, Mr/M, A[vp]/#vp, M/#M]
        & ∀i(1≤i≤alloc&i≠vp → Ar[i]=A[i]));
```

Fig. 8. The formula for verifying the procedure q which is exported from the module declared in Fig. 1. This formula is $H1$ in Fig. 5.

B. Verification Formulae for Modules

The hypothesis $H1$ in Fig. 5 is the requirement for the implementation of procedure q in the module declaration MD in Fig. 1 to be correct. This hypothesis is given in Fig. 8; its first line is the context in which C is the specification of externally defined procedures that are used by the module. The implementation of q may also make use of local procedures that are not exported from the module. These are declared in MDec2, and SLD is their specifications, which is also part of the context.

The first line of the **assume** statement in Fig. 8 states that the module invariant IM and the module correspondence CM are true; the latter essentially defines the value of M in terms of the local module variable lv.

Using the notation described in the last subsection, we denote the input parameters of q by $A[cp]$ and $A[vp]$. Because the specification of q uses cp and vp for this purpose, the former are substituted for the latter. The second line of the **assume** statement in Fig. 8 states that cp and vp are values between 1 and *alloc*, because this is the index range of A. In addition, cp must be different than vp to prevent aliasing. The third line of the **assume** statement is the precondition of q after the above substitution for parameters. The last line of the **assume** statement states that the correspondences and invariants of each instance of AT are true. The former essentially defines the abstract value of each instance in terms of its realization. Because the formal parameters of the correspondence part of AT are as and rs, these are replaced by $A[i]$ and $R[i]$, which are used to denote the abstract value and the realization value of the ith instance of AT. IT refers only to the realization value rs, and hence it is necessary only to substitute for it in IT. This line is an example of where it is necessary to quantify over all of the instances of AT, and how it is achieved with our notation.

The code being verified in Fig. 8 is the body of q. Because new names were substituted for the formal parameters of q in the **assume** statement, the same change of names must be used in verifying its body; specifically, $R[cp]$ and $R[vp]$ are substituted for cp and vp. The reason for this is that the code refers to the realization values of cp and vp, because this is what is actually manipulated by the computer. For example, an assignment to vp in the body of q becomes an assignment to $R[vp]$ after the substitution in *QBody*. On the other hand, the specification of q (e.g., *QPre*) refers only to the abstract values of cp and vp. Of course, the correspondence maps the realization values to the abstract values, and thus provides

the link between the two; but it is important that the names denoting them are used consistently.

The first line of the confirm statement in Fig. 8 states that the module invariant and the invariants of the instances of AT are true. This is necessary because all of these invariants may depend on the values in the local variable lv, which can be changed by the body of q. Of course, q should affect only that part of lv that is used in the realization of vp and M; but this must be verified, which is the purpose of the first line. The first line of the confirm statement also requires that the new realization of vp satisfies IT, because vp is some i between 1 and *alloc*.

The remainder of the confirm statement in Fig. 8 is an implication; its antecedent is the second line of the confirm statement. The purpose of this line is to define the new values of abstract type instances stored in Ar and the new value of the abstract object Mr (see Section VI-A); i.e., the form is "if Ar and Mr give the output values of the abstract objects, then...." But the output values are obtained by mapping the realization values to the abstract space using 1) the correspondence for AT quantified over all of its instances, and 2) the module correspondence. The realization value $R[i]$ of the ith instance of AT is used to define its abstract value $Ar[i]$, because $R[i]$ denotes its value after the execution of q. The remainder of the confirm statement states that the postcondition of q is true, and that each instance of AT has the same value before and after the execution of the body of q, with the exception of vp. The new values of vp and Mr are specified by the postcondition of q. Note that $A[vp]$ denotes the value of vp before executing *QBody*, because A is defined in the **assume** statement, and vp is not changed by the execution of the code, because it is replaced by $R[vp]$ in the code. Thus, $A[vp]$ is substituted for $\#vp$, whose occurrence in *QPost* denotes the value of vp before the execution of q. Similarly, M is substituted for $\#M$, because it denotes the value of M before the execution of q. The other substitutions in the postcondition of q are similar. The last line requires each new value $Ar[i]$ of an instance of AT to be the same as its original value $A[i]$, except for $Ar[vp]$, whose value is specified by *QPost*. This is necessary because q can modify lv, which may be used in the realization of $A[i]$. In fact, sometimes such modifications of the realization data may be desirable for implementation reasons; the only requirement is that the new realization value maps to the same abstract value as the old one. R in the precondition of Fig. 8 refers to input realization values, whereas R in the postcondition refers to output realization values. The postcondition does not refer directly to input realization values, but does refer to the input abstract values.

The verification formula in Fig. 8 is more complicated than the remaining ones, but in many respects is quite similar to the others. For this reason, we described it in considerable detail, and, by the same token, we will describe the others in less detail. The second hypothesis $H2$ in Fig. 5 is the requirement for the implementation of the initialization procedure of AT in the module declaration MD in Fig. 1 to be correct. This hypothesis is given in Fig. 9. Its first line is the context, which is the same as that in Fig. 8. The first line of the assume statement in Fig. 9 states that the module invariant

```
C, SLD \

  assume IM&CM&IPre
    & ∀i(1≤i≤alloc → CT[A[i]/as, R[i]/rs]&IT[R[i]/rs]);

 IBody[R[alloc+1]/rs];

  confirm IM[alloc+1/alloc]& ∀i(1≤i≤alloc+1 → IT[R[i]/rs])
    & (∀i(1≤i≤alloc+1 → CT[Ar[i]/as, R[i]/rs])&CM[Mr/M]
      → IPost[Ar[alloc+1]/as]&M=Mr& ∀i(1≤i≤alloc → Ar[i]=A[i]));
```

Fig. 9. The formula for verifying the initialization of an instance of the abstract type declared in Fig. 1. This formula is $H2$ in Fig. 5.

```
C, SLD \

  assume IM&CM
    & ∀i(1≤i≤alloc → CT[A[i]/as, R[i]/rs]&IT[R[i]/rs]);

  FBody[R[alloc]/rs];

  confirm IM[alloc−1/alloc]& ∀i(1≤i≤alloc−1 → IT[R[i]/rs])
    & (∀i(1≤i≤alloc−1 → CT[Ar[i]/as, R[i]/rs])&CM[Mr/M]
      → M=Mr& ∀i(1≤i≤alloc−1 → Ar[i]=A[i]));
```

Fig. 10. The formula for verifying the finalization of an instance of the abstract type declared in Fig. 1. This formula is $H3$ in Fig. 5.

IM, the module correspondence CM, and the precondition of the initialization are true. The last line of the assume statement states that the correspondence and invariant for each instance of AT are true; this is the same as the last line of the assume statement in Fig. 8.

The code to be verified is the body of the initialization of AT. In this code, rs denotes the new instance of AT being created, and thus is replaced by $R[alloc+1]$, because the range of R is being increased by 1 by adding the new element in position $alloc + 1$. The first line of the confirm statement in Fig. 9 requires the module invariant and the invariant of each instance of AT to be true. Because there is a new instance of AT after the execution of initialization, $alloc+1$ is substituted for *alloc* in IM. Also, the type invariant IT is quantified from 1 to $alloc+1$, because at this point, there are $alloc+1$ instances of AT. The remainder of the confirm statement is quite similar to that in Fig. 8. The only difference in the antecedent is that there are $alloc + 1$ instances of AT instead of *alloc*. *QPost* is replaced by *IPost*, and the $alloc + 1$st instance of AT is substituted for its formal parameter as. The last conjunct of the last line requires each instance of AT, except for the new one, to be unchanged by initialization.

The third hypothesis $H3$ in Fig. 5 is the requirement for the implementation of the finalization procedure of AT in the module declaration MD in Fig. 1 to be correct. The first line of this hypothesis, which is given in Fig. 10, is the context that is the same as that in Fig. 8 and 9. The first line of the assume statement in Fig. 10 states that the module invariant and the module correspondence are true. The last line of the assume statement, which is identical to those in Fig. 8 and 9, states that the invariant and correspondence of each instance of AT are true. The code to be verified is the body of the finalization procedure. Its formal parameter rs is the instance of AT being deallocated. To keep the names consistent in Fig. 10, $R[alloc]$ is substituted for rs in the finalization code. After finialization, there are only $alloc - 1$ instances of AT. The confirm statement in Fig. 10 is almost the same as that in Fig. 9, except that $alloc - 1$ is used in place of $alloc + 1$. The other difference is that there is no postcondition for finalization, because it is transparent at the abstract level.

```
C, SLD \
  assume MPre;
  MBody;
  confirm IM[0/alloc]&(CM → MPost);
```

Fig. 11. The formula for verifying the initialization of the module in Fig. 1. This formula is $H4$ in Fig. 5.

```
P → MPre
C, Sq, SATI \ assume ∃x(P[x/gv]&MPost[x/#gv]); B
------------------------------------------------
C \ assume P; MS; B
```

Fig. 12. The verification rule for module specification MS in Fig. 2. Sq is the specification of procedure q and *SATI* is the specification of the initialization procedure for AT. x is a variable which does not occur in either P or $MPost$.

The final hypothesis $H4$ in Fig. 5 is the requirement for the initialization of the module declared in Fig. 1 to be correct. This hypothesis, which is given in Fig. 11, has the same context as the previous hypotheses. Its assume statement is just the precondition of the module, and the body *MBody* of the module is the code to be verified. The first conjunct of the confirm statement requires the module invariant IM to be true, but 0 is substituted for *alloc*, because there are no instances of AT immediately after the module's declaration. The last conjunct requires the postcondition *MPost* of the module to be true, under the assumption that the module correspondence CM is true. The latter defines the abstract value of M in terms of its realization, whereas *MPost* gives the specification of its initial value.

C. Verification of a Module Specification

An externally defined module can be used by any other module by including the external specification of the former in the latter. The verification rule for the module specifications MS in Fig. 2 is given in Fig. 12.

The bottom line states that the module specification MS followed by a sequence of declaration and program statements is correct. (The last element of B is a confirm statement.) This rule is the same as the one for module declarations (Fig. 6), except that the latter contains an additional hypothesis, its first line, which requires the module declaration to be correct. The reason why this hypothesis is not necessary for module specifications is that the occurrence of a module specification indicates that the module has been defined externally and can be assumed to be correct by any program that uses it. Of course, this assumption can be made because the external declaration of the module must also be verified.

Logically, a module specification belongs in the context of a verification formula. The reason that it occurs in the code is that a module declaration changes the state of the program; e.g., the specification in Fig. 2 is for a module that initializes the value of M and may modify the value of gv. Thus, this specification occurs in the code at the point where this happens. In fact, one purpose of the rule in Fig. 12 is to put the specifications of q and the initialization of AT into the context of the verification of B. The rule also requires the precondition of the module to be true at the point where MS occurs, however, and assumes that the module initialization *(MPost)* is done correctly.

VII. Using the Verification Rules

This section describes the application of the verification rules in the last section to the example in Section IV. This application helps to clarify the notation contained in the rules; also, formulae produced by the rules give insight into our verification method. That is, the reader can judge whether the complexity of these formulae appears to be reasonable or contain unnecessary or missing conditions.

The verification rules are designed to do backward proof by subgoaling, as described in Section VI. If the bottom line of a rule matches the program fragment to be verified, then each hypothesis of the rule yields a condition that must be verified. The main part of this section deals with verifying the implementation *IntegerSet* in Fig. 3, but first a simple example of using integer sets is verified.

Consider the following code, which uses an *IntegerSet*:

$\backslash$*SpecIntSet*; **var** x : *IntegerSet; insert*$(3, x)$;
confirm $x = \{3\}$;

SpecIntSet is the declaration of the abstract or external specification of the module in Fig. 3; Section IV describes how to create *SpecIntSet* from Fig. 3. Applying the verification rule in Fig. 12 reduces the verification problem to the following:

SpecExptProc, SpecInit $\backslash$**var** x : *IntegerSet; insert*$(3, x)$;
confirm $x = \{3\}$,

where *SpecExptProc* is the abstract specification of the exported procedures like *insert*, and *SpecInit* is the specification of the initialization procedure for *IntegerSet*. Essentially, this rule just moves the specifications of the procedures that manipulate integer sets into the context, so that other verification rules have access to them. The first line of the rule creates a subgoal that simplifies to *true*, because both P and *MPre* are *true* by default.

Next the rule in Fig. 7 is applied, which produces the following:

SpecExptProc, SpecInit $\backslash$**assume** $x = \emptyset; insert(3, x)$;
confirm $x = \{3\}$,

because *IntegerSet* does not have a parameter, and P and *IPre* are *true* by default. Again, there is only one nontrivial subgoal, because the first line of the rule simplifies to *true*. Applying the rule for procedure invocation produces the following subgoal:

SpecExptProc, SpecInit $\backslash$**assume** $x = \emptyset$;
confirm $\forall u(u = x \cup \{3\} \rightarrow u = \{3\})$.

This and the remaining rules are not in this paper, but are given in [13]. The quantified u denotes the output value of the integer set, and x denotes its input value. The antecedent of the confirm statement is the postcondition of *insert* after substitutions. Again, there is only one nontrivial subgoal, because the precondition of *insert* is true by default. It is important that the specification of *insert* is part of the context.

```
SE, SA, SC \

  assume IM
    & 1≤x≤alloc
    &InsPre [A [x ]/x ]
    & ∀i (1≤i≤alloc → CIS [A [i ]/as , R [i ]/rs ]&IIS [R [i ]/rs ]);

  InsBody [R [x ]/x ];

  confirm IM& ∀i (1≤i≤alloc → IIS [R [i ]/rs ])
    & (∀i (1≤i≤alloc → CIS [Ar [i ]/as , R [i ]/rs ])
      → InsPost [Ar [x ]/x ,A [x ]/#x ]
        & ∀i (1≤i≤alloc&i≠x → Ar [i ]=A [i ]));
```

Fig. 13. The formula, before substitutions, for verifying procedure *insert* which is exportedfrom the module in Fig. 3. IM is lines 47–51, Fig. 3. *InsPre* is *true* by default. CIS is line 31, Fig. 3. IIS is line 32, Fig. 3. *InsBody* is lines 54–6, Fig. 3. *InsPost* is line 53, Fig. 3. *SE*, *SA* and SC are the specifications of the procedures, *erase*, *add* and *contains*, respectively.

Using the rules for assume statements and confirm statements produces the final verification condition, as follows:

$$x = \emptyset \rightarrow \forall u(u = x \cup \{3\} \rightarrow u = \{3\}),$$

which must be proved by conventional deduction methods. Thus, the verification rules reduce the given verification formula to a single statement in predicate calculus. This reduction depends only on the abstract specification of integer sets, and involves no use of any implementation detail.

To verify the implementation of integer sets in Fig. 3, the verification rule for modules in Fig. 5 is applied to it. This produces nine formulae that must be verified:

- A separate condition for each of the five procedure exported by the module;
- A condition for the initialization of an instance of *IntegerSet*;
- A condition for the finalization of an instance of *IntegerSet*;
- A condition for the initialization of the variables local to the module; and
- A condition for the local procedures not exported from the module.

Since each of these nine conditions can be verified separately by conventional verification techniques, we need to describe only these conditions and how they are produced.

For each procedure exported from a module, there is a condition of the form $H1$ (Fig. 8) that must be verified. Fig. 13 gives the one for the *insert* procedure. The code to be verified (line 6, Fig. 13) is the body of *insert* (lines 54–6, Fig. 3) after substituting $R[x]$ for x, because x is an instance of type *IntegerSet*. The precondition of this code is lines 2–5 of Fig. 13, and its postcondition is the last four lines of Fig. 13. The first line of Fig. 13 is the specification of *erase SE*, the specification of *add SA*, and the specification of *contains SC*, because these local procedures of the module can be invoked by code in the module. Thus, Fig. 13 is just a copy of $H1$ with the generic terms replaced by the details in Fig. 3. We have left the substitutions in Fig. 13 to make its relationship to $H1$ explicit; the result of making these substitutions is given in Fig. 14. Thus, this description of the condition for *insert* has two steps: $H1$ is first converted to Fig. 13, which is then converted to Fig. 14.

The first conjunct of the precondition of Fig. 13 is the module invariant (lines 47–51, Fig. 3). The CM in $H1$ does not occur in Fig. 13, because the module name does not represent an abstract object in this example, and thus there is no module correspondence. Also, there is no constant parameter of type *IntegerSet* to *insert*, which causes the information about *cp* not to occur in Fig. 13. The remainder of the precondition is a direct translation of Fig. 8. The information about CM and *cp* does not occur in the postcondition of Fig. 13 for the same reason. $M = Mr$ does not occur, because the module name does not represent an abstract object in this example. The other parts of the postcondition are a direct translation of the postcondition of $H1$.

```
SE, SA, SC \

  assume IM
    & 1≤x≤alloc
    & ∀j (1≤j≤alloc → A [j ]=rep (R [j ])&distinct (R [j ])& ¬FreeId [R [j ]]);

  begin if not contains (R [x ],i ) then add (i ,R [x ]) end insert ;

  confirm IM& ∀j (1≤j≤alloc → distinct (R [j ])& ¬FreeId [R [j ]])
    & (∀j (1≤j≤alloc → Ar [j ]=rep (R [j ]))
      → Ar [x ]=A [x ]∪{i }
        & ∀j (1≤j≤alloc&j≠x → Ar [j ]=A [j ]));
```

Fig. 14. The formula for verifying procedure *insert* which is the simplified form of Fig. 13. IM is lines 47–51, Fig. 3.

Fig. 13 is somewhat cryptic because of the substitutions in it, but these are removed in Fig. 14, which is much more intelligible. Also, all of the abbreviations have been removed in Fig. 14, except for IM, which was not removed, because of its length. For example, *InsBody* in Fig. 13 was replaced by the code in *insert* in going to Fig. 14. Because the precondition of *insert* is true by default, simplification removes it from Fig. 14. The remainder of the conversion to Fig. 14 is just making the substitutions that occur in Fig. 13. To avoid any possible confusion, the bound i's Fig. 13 were renamed j's in Fig. 14.

It is instructive to translate the formal terms in Fig. 14 into natural language. The names of the different quantities are the key to making this translation: $A[j]$ denotes the abstract input value of the jth instance of type *IntegerSet*, and $Ar[j]$ denotes its abstract output value. $R[j]$ denotes the realization value of the $j\text{th}$ instance of *IntegerSet*. There are *alloc* instances of *IntegerSet*. Because only one abstract type is exported from the module, the *IntegerSet* qualifier can be dropped on A, Ar, R, and *alloc*, which has been done for readability in this section. The first conjunct of the precondition in Fig. 14 is the module invariant, which specifies that each "used" id is the unique name of some *IntegerSet*, and that the used elements of the *mem* array are assigned to these sets. Section IV contains a more detailed description. The second conjunct says that x refers to an *IntegerSet* instance. The last conjunct of the precondition says that for each instance j of type *IntegerSet*, applying the *rep* function to its realization value results in its abstract value. In addition, each element in the set is stored only once in the *mem* array, and the name of the set (which is its realization value) is marked as used in the *FreeId* array.

Line 5 of Fig. 14 is just the code in *insert*; but to keep the names consistent, each x has been replaced by $R[x]$. The first conjunct of the postcondition in Fig. 14 is the module invariant, which must still be satisfied after executing the body of *insert*. The second conjunct specifies that for each instance j of *IntegerSet*, its elements are stored only once in the *mem* array, and its name is marked as used in the *FreeId* array. The

```
SE, SA, SC \

  assume IM&IPre
    & ∀i(1≤i≤alloc → CIS[A[i]/as, R[i]/rs]&IIS[R[i]/rs]);

 IBody[R[alloc+1]/rs];

  confirm IM[alloc+1/alloc]& ∀i(1≤i≤alloc+1 → IIS[R[i]/rs])
    & (∀i(1≤i≤alloc+1 → CIS[Ar[i]/as, R[i]/rs])
      → IPost[Ar[alloc+1]/as]& ∀i(1≤i≤alloc → Ar[i]=A[i]));
```

Fig. 15. The formula, before substitutions, for verifying the initialization of a new instance of *IntegerSet*. IM is lines 47–51, Fig. 3. *IPre* is *true* by default. CIS is line 31,Fig. 3. IIS is line 32, Fig. 3. *IBody* is lines 35–9, Fig. 3. *IPost* is lines 34, Fig. 3.

```
SE, SA, SC \

  assume IM
    & ∀i(1≤i≤alloc → A[i]=rep(R[i])&distinct(R[i])&¬FreeId[R[i]]);

 IBody[R[alloc+1]/rs];

  confirm IM[alloc+1/alloc]& ∀i(1≤i≤alloc+1 → distinct(R[i])&¬FreeId[R[i]])
    & (∀i(1≤i≤alloc+1 → Ar[i]=rep(R[i]))
      → Ar[alloc+1]=∅& ∀i(1≤i≤alloc → Ar[i]=A[i]));
```

Fig. 16. The formula for verifying the initialization of a new instance of *IntegerSet* which is the simplified form of Fig. 15. IM is lines 47–51, Fig. 3. *IBody* is lines 35–9, Fig. 3.

last conjunct is of the form $\alpha \rightarrow \beta$ and γ. The purpose of α is to define $Ar[j]$ to be the output abstract value of the jth instance of *IntegerSet*, which is just the *rep* of its realization value. If this definition holds, then β and γ must be true. β specifies that the output abstract value of x is its input value, with the element i added to the set if necessary. γ specifies that for each instance of *IntegerSet*, except x, its input and output abstract values are the same; i.e., executing *insert* does not cause any side effects in other instances of *IntegerSet*.

Note that *insert* does not change the realization value $R[x]$, but frequently this is not the case, as we shall see. In our notation, $R[j]$ in the precondition of an $H1$ condition refers to the input realization value of instance j, and $R[j]$ in the postcondition refers to its output value. Fig. 14 makes use of this fact because if the body of *insert* had accidentally changed $R[x]$, the postcondition would be violated.

There is a separate verification condition of the form of $H1$ for each of the other four procedures exported from the module. Because these conditions are very similar to those in Fig. 14 and are produced in a similar manner, too, they are not described in this paper. We also do not describe the verification condition for the procedures that are not exported from the module, e.g., *erase*, because they are produced and processed in the same way as ordinary procedures that contain no data abstraction.

The condition for the initialization of a new instance of *IntegerSet* has the form of $H2$ (Fig. 9) and is given in Fig. 15. The first line is the context, which is the specification of *erase, add,* and *contains*, because they are defined, but not exported from the module. Fig. 15 is almost a direct copy of $H2$, using details from this example; e.g., CIS, which represents line 31 in Fig. 3, is used in place of CT. The main difference from Fig. 9 is that the CM and $M = Mr$ in $H2$ do not occur in Fig. 15, because the module name does not represent an abstract object in this example.

Fig. 16 is Fig. 15 after replacing some of the abbreviations and making the indicated substitutions. The precondition in Fig. 16 is the same as the precondition in Fig. 14, except that the second conjunct has been deleted; thus, it has been described above. The new instance of *IntegerSet* being initialized is the $alloc+1$st instance; $A[alloc+1]$ is its abstract value, and $R[alloc+1]$ is its realization value. The instruction of the initialization code on line 38 of Fig. 3 after substitution becomes $R[alloc+1] := i$, which initializes the realization value of the new instance. This is the reason that R is interpreted as a Modula-2 array. In the postcondition of Fig. 16, the R denotes the new array produced by this assignment statement.

```
SE, SA, SC \

  assume IM
    & ∀i(1≤i≤alloc → A[i]=rep(R[i])&distinct(R[i])&¬FreeId[R[i]]);

 erase(R[alloc];
 FreeId[R[alloc]]:=true

 confirm IM[alloc−1/alloc]& ∀i(1≤i≤alloc−1 → distinct(R[i])&¬FreeId[R[i]])
   & (∀i(1≤i≤alloc−1 → Ar[i]=rep(R[i]))
     → ∀i(1≤i≤alloc−1 → Ar[i]=A[i]));
```

Fig. 17. The formula for verifying the finalization of an instance of *IntegerSet*. IM is lines 47–51, Fig. 3.

The first conjunct of the postcondition is that the module invariant is true for all $alloc + 1$ instances of *IntegerSet*. The purpose of the substitution is to include the new instance in the module invariant. The second conjunct specifies that the *IntegerSet* invariant is true for all $alloc + 1$ instances. The second line of the postcondition in Fig. 16 defines Ar to contain the output abstract value of all $alloc+1$ instances. The last line specifies that the abstract value of the new instance is $\emptyset$, and that the abstract values of the other instances are not changed by the code.

The condition for the finalization of an instance of *IntegerSet* is given in Fig. 17. It is produced by putting the details of this example into $H3$, which is given in Fig. 10. Because its production is similar to the condition in Fig. 15, we do not give an intermediate formula, just the final condition. The precondition is the same as the one in Fig. 16. $R[alloc]$ is the instance being finalized. After the code is executed, there will be only $alloc - 1$ instances of *IntegerSet*, and the quantification in the postcondition is over these instances; i.e., $R[alloc]$ is excluded. This is the purpose of the substitution applied to the module invariant. Note that the invocation of *erase* in the code is very important, because, without it, the module invariant might not be true. If the instance being finalized is not empty, the first conjunct of IM (line 47, Fig. 3) is not satisfied unless the memory for storing the elements of the set is released by putting 0's in the appropriate id fields of the *mem* array.

The last verification condition is in Fig. 18; its purpose is to ensure that variables local to the module are initialized correctly. It comes from $H4$ in Fig. 11. Because the module in this example has no pre- or postconditions, Fig. 18 requires the module invariant to be true only after the initialization code is executed. When the module is declared, however, there are no instances of *IntegerSet*, which is the reason that 0 is substituted for *alloc* in the module invariant. This requires all of the id fields of the *mem* array to be 0 and all of the elements of *FreeId* to be true, which is what one would expect.

```
SE, SA, SC \
  MBody;
  confirm IM[0/alloc];
```

Fig. 18. The formula for verifying the initialization of variables local to the module in Fig. 3. IM is lines 47–51, Fig. 3. *MBody* is lines 74–6, Fig. 3.

Each part of each of the above conditions specifies a property of this example that from an intuitive point of view, should be satisfied. In addition, each property that intuitively should be true is embedded in some way in the verification conditions. In this sense, our verification method seems to produce the right kind of verification conditions. At first glance, the conditions seem to be too involved; but in a nontrivial implementation like this, there are lots of details that can go wrong, and the conditions must require that none of this happens. The good news is that the module has to be verified only once. The client modules that declare instances of *IntegerSet* consider them to be abstract sets, and are not aware of the implementation details, but do assume that they have been implemented correctly. Note that the abstract specifications of the exported procedures are very simple, particularly when compared to the above verification conditions, which are full of tricky implementation details.

VIII. Overview of the Metaheory

We have proven that the verification rules in the last section are logically sound and relatively complete in the sense of [7]. To do this, we developed formal semantics for modules, both implementations and specifications, and defined validity in terms of these semantics. Although this development contains a lot of details, it was reasonably straightforward, because our semantics is largely based on the same concepts as the verification rules. For this reason, we give only an overview of the metatheory. Our overview of the metatheory contains much less detail, but has the additional advantage of presenting the key concepts upon which the metatheory is based in a more intuitive way. The reader is referred to [14] for the formal development of the metatheory.

The usual semantics of a procedure's implementation is a function that maps the inputs of the procedure into its outputs. A procedure that manipulates abstract objects cannot be modeled by such a function, because the procedure may produce one of several different outputs for a single input. This is true even when each program statement in the body of the procedure is a function from its input state to its output state, which is somewhat surprising. The reason for this is illustrated in Fig. 19, in which q is a procedure with the input abstract object a_1. There is a function that maps realization values to abstract values; in Fig. 19, both r_1 and r_2, which are realization values, map to a_1; i.e., r_1 and r_2 are two different realizations of a_1. The body of q manipulates these realization values, producing r_3 and r_4 as outputs for r_1 and r_2, respectively, which in turn map to a_3 and a_4. Thus, when the input of q is a_1, its output is either a_3 or a_4, depending on the realization of a_1. Because the purpose of data abstraction is to hide implementation details, we must view q as having one of several possible outputs for the single input a_1.

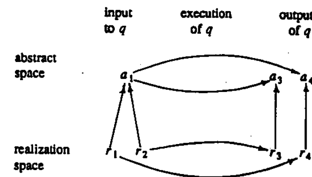


Fig. 19. An example of a procedure which has two outputs for a single input because it manipulates abstract objects.

At first glance, it looks like this procedure is incorrect, because a_3 and a_4 would be the same if the procedure is correct. Indeed, this is normally the case, but there is nothing in Fig. 19 to make the procedure incorrect. In fact, the only way to preclude the case in Fig. 19 is to have the postcondition of q be functional; i.e., for each input, it would be true of only a single output. Normally, postconditions are not required to be functional, and for good reason: They would be overly restrictive. For example, a procedure that computes a minimal cost-spanning tree of a graph should not have a functional postcondition, because, for a single input graph, there may be several different spanning trees with the same minimum cost. The postcondition should not be required to specify which one is the correct output. In addition, if the input graph is represented by adjacency lists, there normally are many different realizations (order of nodes on lists) for the same graph. For one realization, the procedure may output one tree, and for others it may output different trees. Our semantics of procedure implementations considers such a procedure to be correct. But this requires the semantics of the abstract view of procedure implementations to be relations between inputs and outputs. Some of these issues are also addressed by [47].

For procedure q in Fig. 19, both $\langle a_1, a_3 \rangle$ and $\langle a_1, a_4 \rangle$ would be in the semantics of q, which indicates that some executions of q with a_1 as input produce a_3, and others produce a_4. Our semantics for all procedures are relations, because, even though the parameters of a procedure may not be abstract objects, abstract objects may be declared local to the procedure and may be manipulated by its body. In addition, abstractions may be nested, because one abstract object may use others in its realization. Of course, the manipulation of an abstract object in the body of a procedure is a relational operation, which in turn causes the procedure itself to be relational.

Programs, together with their specifications, are interpreted as binary relations between their input environments and their output environments. An environment consists of three kinds of information about the execution of a program. The most obvious part of an environment is the state that maps the names of program variables into their values The assert status is that part of an environment that summarizes the cumulative effect of evaluating the assertions in a program. An implementation must make certain assertions true in order for it to be correct; e.g., the precondition of a procedure must be satisfied before any correct invocation of it. If such an assertion is violated in any execution, the implementation is considered to be incorrect; this is modeled by making the assert status categorically false CF. On the other hand, an implementation

can rely on some assertions being true; e.g., the invocation of a correct procedure will make its postcondition true. If any such assertion is violated in an execution, the assert status becomes vacuously true VT. Once the assert status becomes VT or CF, it stays that way for the remainder of the execution of the program. The remaining assert status is neutral NL, indicating that no assertion in a program has been violated so far.

The third part of an environment gives the meanings of global procedure names by mapping them into their semantics. The semantics of the procedure's implementation (code) is defined by using a minimum fixed-point operation, and in that sense is similar to the denotational semantics of [42]. The semantics of a procedure is a relation between inputs and outputs, however, because there may be several different possible outputs for a single input, as described above. Thus, our semantics is more similar to that in [36], which uses a relation between inputs and outputs defined by a minimum fixed-point operation, instead of the more common function from inputs to outputs. This is the reason that the semantics of a program statement is a binary relation on environments instead of a function from the input environment to the output environment. For example, executing a procedure call in a particular environment can change one of its output parameters to one of several different values, and there are different output environments for these different output values.

The other part of a procedure's semantics is for its specification, which is essentially just the interpretation of both its pre- and postconditions in the semantic space. Most formal semantics for programs do not include such semantics for specifications (e.g., [6]). This is a very important part of our semantics, however, because this is all that is known about an externally defined procedure. That is, the semantics of its implementation is not known, and different environments may contain different implementations for it, which is all right as long as the implementation satisfies its specifications. Thus, the interpretation of the context of a verification formula requires that there be some way to represent a procedure's specification in the semantic space. When such externally defined procedures are called, there must be some representation in the semantic space of whether their pre- and postconditions are satisfied. This is the purpose of the assert status part of an environment. Semantics for modular verification requires the representation for such information, because the implementation of such externally defined procedures is not known; only their specifications are known, and there must be a way to represent the effects of executing such procedures in the output environments of their invocations.

A module declaration has two primary effects: It defines new procedures that can be invoked by the statements following the declaration, and it defines and initializes new variables, one of which may be a single-instance abstract object defined by the module, and perhaps changes the values of some global variables. The interpretation of a module declaration has an output environment that contains these definitions and modifications of the input environment. Some of the procedures exported by the module may have as parameters the abstract object or instances of abstract types defined by the module. These procedures look like any other procedures outside the module, however, because the outside world knows only about the abstract view of these objects, not about the way in which they are represented and manipulated inside the module.

For the most part, the semantics of a module are the collective semantics of the procedures defined by the module. As with ordinary procedure declarations, their interpretation produces a categorically false output environment if some of their implementations do not satisfy their specifications. The interpretation of these procedures is very different, however, if they manipulate the realization of data abstractions defined by the module, and this is the primary new thing in the semantics of modules.

Consider the declaration of procedure q in Fig. 1. There are actually two different semantics for q: The abstract view, used by code outside the module, in no way depends upon the realization of the abstract objects that are parameters to q. But inside the module, the implementation of q is viewed as code that processes the realization data of the abstract parameters of q. The requirement for the declaration of q to be valid is that the implementation of q correctly manipulates the realization data. This has three parts: The module invariant and the abstract type invariant must be maintained. Second, the implementation cannot produce side effects in abstract objects that are not parameters to q. The third condition is that the parameters of q are correctly manipulated, which is specified by the pre- and postcondition of q. These are abstract specifications, however, and must be transformed into realization conditions, which is done using the module correspondence CM and the abstract type correspondence CT. The verification formula for the declaration of q in Fig. 8 is based on the same three requirements as described in Section VI. Since the semantic notion of correctness (validity) and the axiomatic notion have the same basis, the proof of soundness turns out to be relatively straightforward. However, the proof does entail a considerable amount of detail, which is given in [14].

The validity of a module declaration also requires the semantic version of the verification conditions for the initialization and finalization of AT (Figs. 9 and 10) to be true, as well as the condition for the initialization of the module (Fig. 11). The latter assumes that the module precondition is satisfied.

To use an externally defined module, the module specification without implementation must occur in the code at the appropriate place. The validity of this code requires that the semantic version of the module precondition is true in the input environment of the interpretation of the module specification. Because the focus of this research is on modular verification, we formally define only the validity of program modules, not the validity of entire programs. A valid program is a collection of valid modules in which each item imported by a module is exported by precisely one other module, and in which the specifications of imported and exported items are compatible. The verification of code that uses a module as a client is based on the abstract specification of the module, and this is

the way that the interdependency of modules on one another is taken care of by the verification process. Thus, an entire program is verified by verifying each module independently, which is the reason that the main conceptual issue of this research is the validity of individual modules rather than entire programs.

Our definition of validity is significantly different from the more usual one, e.g., [6], because it depends on assertions internal to the program. For example, a program is invalid if the precondition of a module is not true when it is declared. Similarly, valid code requires the preconditions of procedures to be true before they are invoked. The more usual definition of validity depends only on the pre- and postconditions of an entire program, not on any internal assertions.

We have proven that the verification rules for modules in Section VI are logically sound.

Theorem 1: If a module can be proven to be correct by the verification rules for modules in Section VI, together with a soundproof system for other programming constructs and the specification language, then the module is valid with respect to the semantics outlined above.

The essential part of the proof is that each verification rule in Section VI preserves validity; i.e., if each hypothesis of a rule is valid, then it can be proven that the bottom line of the rule must also be valid.

We have also proven that the verification rules are relatively complete.

Theorem 2: If a module is valid as outlined above, then it can be proven to be correct by the verification rules in Section VI, together with a logically complete deduction system for other programming constructs and the specification language.

Since the specification language contains number theory, there is no effective complete deduction system for it. This is the relative completeness concept of [7]. Theorem 2 is a way to state that any incompleteness in a deduction system that uses our verification rules for modules comes from the incompleteness of the deduction system for our specification language, which contains number theory, and not from our verification rules, because there are relatively complete verification rules for the other programming constructs in Modula-2.

The proof of Theorem 2 requires the specification language to be expressive, which means that the strongest postcondition of any sequence of program statements can be expressed in the specification language. If we exclude the use of real numbers in Modula-2, our specification language is expressive, because it contains both first-order logic and number theory. Oppen and Cook [41] have proven that such a specification language is expressive for a programming language that manipulates the kind of data structures found in Modula-2. In addition, if the strings in [43] are added to the specification language, it is expressive for a large class of data types as shown in [12]. The other essential part of the proof of Theorem 2 shows that the verification rules for modules preserve validity in the reverse direction. That is, each hypothesis of a verification rule is valid if the bottom line of the verification rule is valid. The proof of this is quite straightforward but intricate, because the semantics of modules is based on the same set of concepts as the verification rules.

IX. Conclusion

This paper presents a method for the modular specification and verification of data abstractions in which multiple abstract objects share a common realization level data structure. This kind of data abstraction is important because it provides for efficient use of memory; i.e., it allows the amount of memory allocated to the realization of an abstract object to be dynamic, so that only the amount of memory needed for its realization is allocated to it at any one time. Section III is a high-level description of this kind of implementation and the kind of problems that must be dealt with by its specification and verification. To be explicit, an example of this kind of data abstraction is given in Section V. Although a number of programming languages provide good support for this kind of data abstraction, there has been limited research on its specification and verification.

An important property of our method is that it allows data abstractions to be dealt with modularly. Each data abstraction can be specified and verified individually. Its abstract specification is made available for use by other program modules, but all of its implementation details are hidden, which simplifies the verification of code that uses the abstraction. Although most man–machine verification systems use modular methods, the methods in most theoretical investigations do not, because they do not provide for externally defined procedures whose specifications are known to external users, but whose implementations are hidden. This has a large impact on the semantics of programs, because the effect of invoking an externally defined procedure is determined by its specification, not by its implementation. This must be reflected in the definition of a correct (valid) program, and the definitions in the literature, e.g., [6], do not provide for this.

Our method requires that specifications be inserted into the code at appropriate places, and we have developed semantics for this mixture of program statements and specifications. These semantics provide for modularity, because implementations are defined to be correct (valid) with respect to the specification of externally defined procedures and data abstractions whose implementations are unknown. With this semantics, our verification method has been proven to be logically sound and relatively complete in the sense of [7]. In addition to the verification of shared realizations, the other new result in this paper is that our verification method for data abstraction is both modular and relatively complete. As far as we know, the literature does not contain a modular verification method based on abstract models [27] that has been proven to be relatively complete.

The use of shared realizations impacts specification and verification in several related ways. The manipulation of one abstract object may inadvertently produce side effects in another abstract object. Without shared realizations, such unwanted side effects can be prevented by scoping rules, but this is not possible with shared realizations. Instead the absence of such side effects must be explicitly proven by the verification method. Thus, the flexibility provided by shared realizations shows up as additional requirements that must be verified. These are primarily specified by invariant

properties of the implementation, which in some cases require quantification over the instances of an abstract type that are currently active (allocated). An example is given that supports this claim. Our specification language provides for such quantification, which cannot be done in most specification languages. The verification conditions generated by our method contain a significant amount of such quantification. For example, many of the verification conditions require that *for all instances of an abstract type*, except the ones being manipulated, their abstract values are unchanged by the code doing the manipulation. Although such conditions are not required for verifying abstractions without shared realizations, the complexity of the verification conditions produced by our method seems to be reasonable, as illustrated by an example in this paper. That is, each part of each condition specifies a property of the example that intuitively should be true of a correct implementation. In addition, each property that intuitively should be true is embedded in some way in the verification conditions.

Modula-2 is the programming language used in this paper to describe our method, but it is directly applicable to other languages, such as Ada, whose abstraction mechanisms are based on exporting types from a module. The relationship between data abstraction and advanced programming techniques like concurrency and inheritance, however, are beyond the scope of this paper. Data abstraction is obviously very important in object-oriented programming and is also important in some software systems that contain concurrency. Shared realizations are a useful implementation technique for data abstraction, and thus can be beneficially used in some such systems. At least some of the basic concepts underlying our method should be applicable to these systems. One of the issues that must be addressed is modularity, so that any component of a system can be verified, given only the specifications, but no implementation details, of the other components of the system that it uses. Although our method is modular for its domain of applicability, more research is needed to determine how it can be integrated with specification and verification methods for concurrency and inheritance while maintaining modularity.

Acknowledgment

We want to thank the referees for suggesting a number of ways to improve this document. Several of these greatly enhanced the readability of the paper. For example, Section III was added at their suggestion to provide a high-level overview of the problem and our approach to solving it. They also made some excellent suggestions for reorganizing the presentation of some of the technical material. We are grateful to the referees for these suggestions and for the amount of time they spent in carefully reviewing the document.

References

[1] U.S. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL–STD–1815A–1983, Dec. 1983.
[2] R. J. R. Back, "A calculus for refinements for program derivations," *Acta Informatica*, vol. 25, pp. 593–624, 1988.
[3] B. Belkouche and J. E. Urban, "Direct implementation of abstract data types from abstract specifications," *IEEE Trans. Software Eng.*, vol. SE-12, no. 5, pp. 649–661, May 1986.
[4] J. A. Bergstra, J. Heering, and P. Klint, *Algebraic Specification.* Reading, MA: Addison-Wesley, 1989.
[5] E. Chang, N. E. Kaden, and W. D. Elliott, "Abstract data types in Euclid," *SIGPLAN Notices*, vol. 13, no. 3, pp. 34–42, Mar. 1978.
[6] E. M. Clarke, "Programming language constructs for which it is impossible to obtain good Hoare axiom systems," *J. ACM,*, vol. 26, no. 1, pp. 129–147, Jan. 1979.
[7] S. A. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM J. Computing,*, vol. 7, pp. 70–90, Feb. 1978.
[8] O. J. Dahl, B. Myhrhaug, and K. Nygaard, "The SIMULA 67 common base language," Pub. S-22, Norwegian Computing Center, Oslo, Norway, 1970.
[9] J. E. Donahue, "Complementary definitions of programming language semantics," Tech. Rep. CSRG-62, Comput. Syst. Res. Group, Univ. of Toronto, ON, Canada, Nov. 1975.
[10] H. Ehrig, "Algebraic implementation of abstract data types: Concept, syntax, semantics and correctness," *Lecture Notes in Computer Science: Automata, Languages and Programming*, vol. 85, pp. 142–156, 1980.
[11] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science, Vol. 6. New York: Springer-Verlag, 1985.
[12] G. W. Ernst, R. J. Hookway, J. A. Menegay, and W. F. Ogden, "Modular verification of Ada generics," *Comput. Languages*, vol. 16, pp. 259–280, 1991.
[13] ——, "Semantics of programming languages for modular verification," Tech. Rep. CES-85-4, Dept. Comput. Eng. and Sci., Case Western Reserve Univ., Cleveland, OH, 1985.
[14] G. W. Ernst, R. J. Hookway, and W. F. Ogden, "Modular verification of data abstraction in Modula-2," Tech. Rep. CES-89-10, Dept. of Comput. Eng. and Sci., Case Western Reserve Univ., Cleveland, OH, 1989.
[15] G. W. Ernst and W. F. Ogden, "Specification of abstract data types in Modula," *ACM Trans. Programming Languages Syst.*, vol. 2, no. 4, pp. 522–534, Oct. 1980.
[16] J. D. Gannon, R. G. Hamlet, and H. D. Mills, "Theory of modules," *IEEE Trans. Software Eng.*, vol. 14, no. 7, pp. 820–829, July 1987.
[17] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile, "An overview of AFFIRM: A specification and verification system," *Inform. Processing.* Amsterdam: North-Holland, pp. 343–347, 1980.
[18] C. M. Geschke, J. H. Morris, and E. H. Satterwaite, "Early experience with Mesa," *Commun. ACM*, vol. 20, no. 8, pp. 540–553, Aug. 1977.
[19] A. Goldberg and D. Robson, *Smalltalk-80: The Language*, Reading, MA: Addison-Wesley, 1989.
[20] G. A. Gorelick, "A complete axiomatic system for proving assertions about recursive and non-recursive programs," Tech. Rep. 75, Dept. of Comput. Sci., Univ. of Toronto, ON, Feb. 1975.
[21] J. Grabowski and P. Lescanne, "Modular algebraic specifications," *Mathematical Research: Algebraic and Logic Programming*, Berlin: Akademie-Verlag, 1989, pp. 168–179.
[22] D. Gries and D. Volpano, "The transform: A new language construct," *Structured Programming*, vol. 11, pp. 1–10, 1990.
[23] J. Guttag, J. J. Horning and J. M. Wing, "The Larch family of specification languages," *IEEE Software*, vol. 2, no. 5, pp. 24–36, 1985.
[24] J. V. Guttag, E. Horowitz and D. R. Musser, "Abstract data types and software validation," *Commun. ACM*, vol. 21, no. 12, pp. 27–52, Dec. 1978.
[25] B. Hailpern and S. S. Owicki, "Modular verification of concurrent programs," *Symp. Principles Programming Languages*, 1982, pp. 322–336.
[26] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–581, 1969.
[27] ——, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, pp. 271–281, 1972.
[28] S. Igarashi, R. L. London, and D. Luckham, "Automatic program verification I: A logical basis and its implementation," *Acta Informatica*, vol. 4, pp. 145–182, 1975.
[29] P. Jalote, "Synthesizing implementations of abstract data types from axiomatic specifications," *Software: Practice and Experience*, vol. 17, no. 11, pp. 847–858, Nov. 1987.
[30] R. A. Kemmerer, "A critique of the Gypsy verification system," *Verification Assessment Study, Final Report, The Gypsy System*, Vol. II, R. A. Kemmerer, Ed. Fort George G. Meade, MD: National Computer Security Center, 1986, pp. 49–61.
[31] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek, "Report on the programming language Euclid," *SIGPLAN Notices*, vol. 12, no. 2, pp. 1–79, Feb. 1977.
[32] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* Cambridge, MA: MIT, 1986.
[33] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek, "Proof rules for the programming language Euclid,"

Acta Informatica, vol. 10, pp. 1–26, Jan. 1978.

[34] D.C. Luchkam and W. Polak, "A practical method of documenting and verifying Ada programs with packages," *Proc. ACM-SIGPLAN Symp. Ada Programming Language*, 1980, pp. 113–122.

[35] D.C. Luckham, F.W. Von Henke, B. Krieg-Brueckner, and O. Owe, *ANNA: A Language for Annotating Ada Programs*, Ref. Man., *Lecture Notes in Computer Science 260*. New York: Springer-Verlag, 1987.

[36] G. Nelson, "A generalization of Dijkstra's calculus," *ACM Trans. Programming Languages Syst.*, vol. 11, pp. 517–561, Oct. 1989.

[37] B. Meyer, *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[38] J.M. Morris, "Laws of data refinement," *Acta Informatica*, vol. 26, pp. 287–308, 1989.

[39] D.R. Musser, "On proving inductive properties of abstract data types," *3rd ACM Symp. Principles of Programming Languages*, 1980, pp. 154–162.

[40] E.R. Olderog, "Sound and complete Hoare-like calculi based on copy rules," *Acta Informatica*, vol. 16, pp. 161–197, 1981.

[41] D.C. Oppen and S.A. Cook, "Proving assertions about programs that manipulate data structures," *Proc. 7th Ann. ACM Symp. Theory of Computing*, 1975, pp. 107–116.

[42] D. Scott and C. Strachey, "Toward a mathematical semantics for computer languages," *Computers and Automation*. New York: Wiley, 1972, pp. 19–46.

[43] D. Scott, "Some definitional suggestions for automata theory," *J. Comput. Syst. Sci.*, vol. 1, pp. 187–212, 1967.

[44] J. Spitzen and B. Wegbreit, "The verification and synthesis of data structures," *Acta Informatica*, vol. 4, pp. 127–144, 1975.

[45] A. Stoughton, "Substitution revisited," *Theoretical Comput. Sci.*, vol. 59, pp. 317–325, 1988.

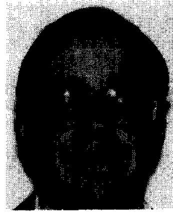
[46] B. Stroustrup, *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.

[47] M. Wand, "Specifications, models, and implementation of data structures," *Theoretical Comput. Sci.*, vol. 20, pp. 3–32, 1982.

[48] N. Wirth, *Programming in Modula-2*, 3rd Ed. New York: Springer-Verlag, 1985.

[49] W.A. Wulf, R.L. London, and M. Shaw, "An introduction to the construction and verification of Alphard programs," *IEEE Trans. Software Eng.*, vol. 2, no. 12, pp. 253–264, Dec. 1976.

[50] W.D. Young and D.I. Good, "Generics and verification in Ada," *Proc. ACM-SIGPLAN Symp. Ada Programming Language*, 1980, pp. 123–127.

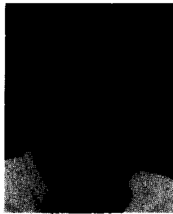


George Ernst received the B.S., an M.S., and a Ph.D. degrees in electrical engineering from Carnegie Institute of Technology, Pittsburgh, PA.

Since his graduate studies, he has been on the faculty of Case Western Reserve University, Cleveland, OH, where he currently is an Associate Professor in the Department of Computer Engineering and Science. His main research interests are artificial intelligence and the specification and verification of software systems.

Ray Hookway received the B.S., M.S., and Ph.D. degrees from Case Western Reserve University, Cleveland, OH.

He is a Software Engineering Manager at Digital Equipment Corp., Acton, MA. Prior to joining DEC, he was Director of Engineering at ENDOT, Inc., in Cleveland, OH, and was a faculty member in the Department of Computer Engineering and Science, Case Western Reserve University. His interests include, compilers, simulation, and formal verification.



William F. Ogden received the B.S. degree from the University of Arkansas, and his M.S. and Ph.D. degrees are from Stanford University, Stanford, CA.

He is an Associate Professor of Computer and Information Science at Ohio State University, Columbus, OH. Previously, he served on the faculties at Case Western Reserve University and the University of Michigan. His research interests include software reuse, software specification, and program verification.