GENERICS AND VERIFICATION IN ADA

William D. Young
Donald L. Good

ICSCA-ADA-2
(ICSCA-CMP-19)

December, 1980


Institute for Computing Science and Computer Applications
Certifiable Minicomputer Project
The University of Texas at Austin
Austin, Texas 78712
TABLE OF CONTENTS

1 Introduction.............................................. 2
2 Related Work.............................................. 2
3 The Goal of Generic Verification......................... 3
  3.1 Proof Modularity..................................... 4
  3.2 Proof Rules........................................... 4
  3.3 Specification Capabilities............................ 6
4 Generic Verifiability in ADA............................. 7
5 Conclusions............................................... 11
ABSTRACT

This paper explores the restrictions a mechanism in the style of the Ada generics facility would have to satisfy in order to be amenable to existing verification techniques. "Generic verification" is defined and defended as the appropriate goal for any such facility. Criteria are developed for generic verification to be possible and then Ada is evaluated with respect to these criteria. An example of the application of these techniques to an Ada unit is presented to show that generic verification is possible at least on a subclass of Ada generic units. Finally some potential applications of verified generic units are presented.
1 Introduction

One of the more novel features of the Ada programming language is the ability to define generic program units. This facility permits translation time parameterization of subprograms and modules to prevent replication of text, yield increased readability, and encourage functional and data abstraction. Because of its relative novelty, this mechanism needs to be examined in the light of current programming language research.

A question which has already begun to be addressed with respect to Ada is the following: which features of the language are amenable to the techniques of program verification? In the context of a verification analysis being carried out at the University of Texas [11], the question arose as to the relationship of generics to verifiability. This paper attempts to explore this relationship for Ada.

This paper is divided into five sections. In section 2 we examine the generics facility of Ada in relation to similar facilities for other languages. Sections 3 and 4 describe requisite features for generic verification and examine the Ada facility with respect to these features. Finally, in Section 5 conclusions of this research are presented and some benefits of verified generic units are given. This presentation assumes familiarity with the Ada generics facility [4,10] as well as with techniques of program verification.

2 Related Work

Generic unit definition facilities in the style of Ada are not a common feature among programming languages. The facility bears some similarity, however, to "syntax macros" [1] which have been available for quite some time in so-called extensible languages. Ada attempts to temper generality with implementability in the generics mechanism. Therefore it is significantly more constrained than a general text substitution facility.

A very general generics mechanism is present in the CLU programming language [12] but in the context of a dynamic typing scheme. CLU [6] permits modules parameterized by types. In addition there is a mechanism in CLU for specifying constraints on the generic parameters via a where clause. Euclid [5]
provides parameterization by types but requires that potential parameter values be enumerated in advance. Alphard [14] forms are a quite general generics mechanism. As with Clu there is the ability to restrict the generic parameters by specifying required characteristics. Various other languages have facilities similar to these and are cited in [4].

None of these languages except Euclid and Alphard has been particularly concerned with program verification issues. The Euclid generics facility does not present a particular problem for verification since all possible values of the generic parameters are indicated explicitly. Alphard, then, is the only significant attempt to apply verification techniques to a fairly general generics facility. Other languages which have been designed with verification in mind, e.g. Gypsy [2], have not included such facilities.

3 The Goal of Generic Verification

In the simplest analysis, generics have no necessary relationship to program verification. The generics facility is essentially a context sensitive macro facility and any verification effort can be directed toward the generic instantiations. Given that the generic program unit is merely a template for its instantiations and in no way affects the program state, this strategy is both sound and complete with respect to generics. However, there are potentially a large number of instantiations for each unit. If each of these must be proven individually then not only will the proof be correspondingly more complex, but also the benefits of modularity, "factorization", and functional abstraction gained at the program design level will have no verification time counterparts. One would like to do better.

The goal of the verification of generic units should be, quite simply, "generic verification". By this is meant the ability to verify a program unit once, in generic form, and have a "generic proof" applicable to each instantiation of the unit. A facility which permits this imports the advantages of modularity into the proof domain and reduces the complexity of the proof process significantly.

Verification at the generic level presupposes an underlying proof apparatus for the language. Though there is none presently for Ada, it is almost a certainty that such an apparatus will be defined. Studies are already underway to determine the amenability of various Ada constructs to verification techniques.
[7, 11]. The verifiability of a significant subset appears likely. An extension to generic verification, however, requires, in addition to the verifiability of the individual non-generic constructs of the language, that the following hold:

1. Proof modularity must be possible at the level of program units;
2. The proof rules for the language must be sufficiently general to allow generic proofs to be carried out; and,
3. The specification techniques defined for the language must be adequate to specify generic properties.

Each of these points will now be examined in some detail.

3.1 Proof Modularity

The possibility of carrying out generic verification presupposes that there is already proof modularity at the level of (non-generic) program units. That is, given a particular unit it must be possible to prove its correctness independently of the environment in which it is called. Without this capability, generic verification, which requires proof modularization at an even higher level of abstraction, is impossible. Languages such as Gypsy [2] which are designed with verification issues in mind take care to preserve this sort of proof modularity by restricting non-local referencing, relying on abstract data types, incorporating abstract specification facilities, etc. Ada has various classes of program units—functions, value returning procedures, procedures—which permit proof modularity to varying degrees. A discussion of this question is presented elsewhere [11]. It is assumed for this discussion that such proof modularity is possible for the Ada program units considered.

3.2 Proof Rules

Proof rules are usually applicable only to a single type or operation or to a very limited set of types or operations. However, generic verification requires proof rules which are parameterized in such a way that they capture the entire range of semantic variance possible for a statement within a generic unit.

Consider the following ostensibly general rule in Hoare's [3] notation:
The validity of this rule relies upon the transitivity of "=". However, in a language such as Ada which permits overloading of infix operators this in turn depends upon the prevailing definition of "=". (This supposes that such overloading is effective within the specification language, as it almost certainly must be.) A more "generic" form of this rule is:

\[(1) \{x*y \land y*z\} \text{ null } \{x*z\}\]

where \(\text{Eq-Rel("*")}\) specifies that "*" denotes an equivalence relation on its parameters. (Transitivity would suffice).

Note that (1) is an instantiation of (2). The difference in generality between these rules depends upon the predicate \(\text{Eq-Rel}\). That is, \(\text{Eq-Rel}\) defines the class of operations for which the generic proof rule can be instantiated (or for which it is of interest to do so). Call such predicates "qualifiers". Qualifiers can be arbitrarily complex and can even be independently defined specification functions. In ADA-like syntax, \(\text{Eq-Rel}\), for instance, might be defined as follows:

```
Generic (type T;
  function "*" (a, b:T) return boolean;

Function Eq-Rel ("*") return boolean is
begin
  result := for all a, b, c in T:
             (a * a;
              a*b => b*a;
              a*b and b*c => a*c);
end Eq-Rel;
```

Such specification functions, of course, are intended as proof-time constructs which are not compiled or executed. Some such definitional mechanism for selecting classes of operators
and types is a necessary part of the generic specification language since few useful inferences can be drawn which are applicable to all operators or all types. Hence, to define truly useful generic rules it must be possible to provide some explicit characterization of their range of applicability. Qualifiers serve this function.

On the other hand, it is possible to qualify a proof rule so completely that it is applicable to only a single operator or type. Qualification should not constrain the applicability of a proof rule to such an extent that its generic character is lost. Ideally, a qualifier will define a useful and natural superclass of the class of operators or types for which the generic unit being proven can be instantiated. More realistically it will define a significant subclass of the class of operations or types for which it will be instantiated.

Fortunately, the degree of generality required for parameters to generic units is invariably less than total. The rules necessary to prove a generic integration unit with type parameter \( T \), for instance, could probably all be qualified by Numeric (T). An attempt to instantiate the unit with \( T \) as a character string type then could either be interpreted as an error or generate a legality assertion in the style of Euclid [13] that the unit was unproven for such actual parameters.

3.3 Specification Capabilities

The usual techniques of program verification require the ability to supply a specification of the intended effects and performance of a program segment which is independent of the program text. Therefore, specification facilities are necessary either within the programming language or as an adjunct to the language. Moreover, the specification of a generic program unit must be parametric to the same extent that the unit is. Otherwise the specifications may not capture the full generality of the corresponding unit and hence may not be applicable to certain values of the generic parameters. It may not be possible to supply such general specifications because the range of the generic parameters may be too great to admit a useful characterization of the unit's behavior under all possible instantiations. If this is the case the specifications may need to contain qualifiers similar to those we have described for the proof rules. It is assumed that the unit will be instantiated only for some well defined subset of possible parameters. The specifications, then, apply to this subset and the proof is applicable only to instances which satisfy these assumptions.
4 Generic Verifiability in ADA

Modules, in turn, are either packages or tasks. Our discussion will concern only generic subprograms and packages. For the verification of tasks, the complexity introduced by concurrency overshadows that introduced by generics and must be managed first.

The Ada generics facility is described as a "context sensitive macro facility" [4, p. 132]. Macros in general hinder modular verification because they permit inline substitutions which essentially generate new programs. A simple substitution of variable names, for instance, may result in overloading which hides previous declarations and alters the meaning of a program substantially. This may entirely invalidate previous verification efforts. The definition of generic subprograms or packages in Ada, however, is not the sort of unrestricted macro facility which permits inline substitutions of arbitrary pieces of text. The parametric substitutions are constrained by the declarations of the formal generic parameters. But are they sufficiently constrained to permit generic verification?

If used in its full generality, the generic facility probably does not admit generic verification. Consider the following Ada subprogram definition:

```
Generic (type T;
   procedure A (x,y: in out T))

procedure P (x,y: in out T);
begin
   A (y, x);
end P;
```

P is a template for any operation which can be defined on two parameters of any type. The possible range of parametric substitutions is so large that no useful generic assertion can be made. It must be possible to qualify the parameters sufficiently that useful generic specifications can be formulated and generic proof rules applied.

There is some justification from purely design considerations to think that generic units should be written in such a way that this is possible. Procedure P above, for
instance, is too general to really be acceptable or useful. As
the Ada designers themselves point out [4, p. 2-5] "language
constructs which do not express similar ideas should not look
similar". A unit SWAP should not in one instantiation
interchange the values of two data locations and in another
instantiation solve partial differential equations. Generic
units should have properties which are inherited by all
1
instantiations. However, Ada imposes no language constraints in
this regard.

Despite this negative evaluation it may be possible to carry
out generic verification in Ada on a limited scale. On one hand,
some units are inherently generic. Most formulations of stacks,
for instance, have this property. The functionality of the unit
does not really depend upon the type of the items stacked—that
is, upon the generic parameter. This property seems to be
related to the ability to provide algebraic specifications for
the unit [7]. Most generic units without subprogram parameters
probably fit into this category. On the other hand, for many
units it is possible to qualify the parameters in the
specifications sufficiently that generic proof rules can be
applied. This tactic will probably be necessary if subprogram
parameters enter in any essential way into the generic
definition. For instance, an Ada binary search routine with
header

Generic (function "<" (a, b: node_label)
   return boolean);

function binary_search (root: tree;
   key: node_label) return tree is

assumes a binary tree (some previously defined type) ordered with

1
This is analogous to the problem of unrestrained overloading.
"=" should denote an equivalence relation for which
substitutivity holds regardless of the types of the operands. If
this is true then the generic axioms of equality can be used in
proving any program segment within which "=" appears.
respect to the "<" relation. Moreover, for most reasonable instantiations of the unit, "<" will be at least a partial order over node labels. In verifying the unit then one can rely upon axioms of partial orders (and probably would have to in order to prove anything of interest about the program). The programmer concerned with writing verifiable software can ensure that the units he writes are either inherently generic or can be qualified in the requisite fashion.

An alternative possibility is to install within Ada a facility to make it possible to constrain the generic parameters. This is the approach of CLu [6]: a \texttt{where} clause specifies properties of generic type parameters including required operations on that type. For Ada this could take the form of a constraint to classes of types, subprograms, etc. Thus, instead of writing

\begin{verbatim}
generic (T: type)
\end{verbatim}

and then providing a proof of the unit applicable only over the intended range of instantiation— the numeric types, for instance—it would be possible to write

\begin{verbatim}
generic (T: Numeric type)
\end{verbatim}

This permits syntactic specification of a level of parameterization intermediate between total generality and constraint to a single value. We now consider these concepts with regard to the verification of Ada generic units.

The utility of generic program units without subprogram parameters is quite limited. For objects belonging to a type which is a generic parameter the only available operations within the unit are assignment and equality comparison. Even these are denied if the unit is declared to be "restricted". However, it is exactly these limitations which insure that most generic units without subprogram parameters will probably be susceptible to generic verification. Assignment and equality comparison (if equality is "standard") have a set of axiomatic properties which are largely independent of the types of the operands. These can
be directly utilized in carrying out the generic proof.

The addition of subprogram generic parameters introduces considerable additional complexity into the problem of verification. For verification to succeed it must be possible to provide specifications for each generic subprogram parameter. The generality of Ada is a hindrance here. The only information available from the generic unit definition is the header, which may itself depend upon other generic parameters. What is needed is an input-output specification for the class of subprograms which might reasonably be supplied as actual parameters at instantiation. This class will likely be restricted by qualifiers within the entry specification for the unit and so will be a proper subset of the class of functions (or procedures) definable with that header. The size of this subset depends upon the degree of qualification and probably bears some relation of inverse proportionality to the success of the verification attempt.

Illustrative of these concepts is the following annotated generic bubblesort routine which orders an array B according to some binary relation L. Assertions are enclosed in { }. 

Generic (type item;
     function L(a1, a2: item) return boolean)

Procedure Bubble_sort
   (B: in out array (1..n) of item) is
begin
   for i in 1..n loop
      for j in 1..n loop
         {for all k in [1..i-1]: Sorted (1, k)}
         for all k in [i+1..n]: L(B(i), B(k))
         if L(B(j), B(i)) then
            interchange (B(i), B(j));
         end if;
         {for all k in [i+1..j]: L(B(i), B(k))}
      end loop;
     {for all k in [1..i]: Sorted (1, k)}
   end loop;
   {Sorted (1, n)}
end Bubble_sort;

The exit specification for the routine is "Sorted (1, n)" where
Sorted \((i, j) = \)

for all \(k, l\) in \([i..j]::\)

\((k \geq 1) \rightarrow L(B(k), B(l)))\)

The proof is straightforward except that proving the inner loop requires proving

\{-L(A(j), B(1)) and for all \(k\) in \([i+1..j-1]::\)

\(L(B(1), B(k)))\}

interchange \((B(i), B(j))\);

\{for all \(k\) in \([i+1..j]::\) \(L(B(i), B(k)))\}

which is not true for an arbitrary binary relation \(L\), though it is true if \(L\) is transitive. For most instantiations \(L\) will be a transitive relation.

As in many other instances, the routine is not provable in its given generality. Suitably qualified, however, the proof is applicable to most of the useful and desirable instantiations.

5 Conclusions

In this paper an attempt has been made to address the following two questions. What is necessary to apply the techniques of program verification to generic program units? Can these techniques be applied to generics as defined in Ada? This analysis has been mainly negative with respect to Ada—the generics facility is simply too general to hope for verification of arbitrary generic units. However, if the programmer defines units which are generic on a somewhat lesser scale, then these techniques may apply.

For verification on large software projects, generic verification is certainly worth the effort. The following applications suggest themselves:

1. A completely verified generic unit can be reliably instantiated any number of times without re-verification of any of the instantiations.
2. As has been indicated, it may be feasible to verify some generic units for only a proper subset of possible instantiations. It may be reasonable to maintain a database of Ada generic units with information on the classes of parameters for which they have been verified.

3. An incremental development system in the style of [9] might be developed to automatically evaluate the additional work necessary to verify a particular instantiation.

4. Given a database of verified generic units, programmers could be encouraged to instantiate these units rather than define similar non-generic units which would require additional verification.

5. Finally, given a group of non-generic units, it may be possible to determine that certain of them are actually instances of verified generic units or functionally equivalent to such units.

These applications indicate that the entire area of generics and verification certainly merits additional study.

Acknowledgements:

This work was supported by the Defense Advanced Research Projects Agency under contract No. N00039-76-C-0080. We gratefully acknowledge the assistance of Anand Tripathi and Richard M. Cohen.
REFERENCES


