A PRELIMINARY EVALUATION
OF VERIFIABILITY IN ADA

Anand R. Tripathi
William D. Young
Donald I. Good

November, 1980

Published in the Proceedings of the ACM National
Conference, Nashville, October, 1980.

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>2</td>
</tr>
<tr>
<td>2 Verifiability and Language Features</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Modularization of Proofs</td>
<td>4</td>
</tr>
<tr>
<td>2.2 The Privacy Principle</td>
<td>4</td>
</tr>
<tr>
<td>3 Data Types in Ada</td>
<td>5</td>
</tr>
<tr>
<td>3.1 The Approximate Types</td>
<td>5</td>
</tr>
<tr>
<td>3.2 Access Types</td>
<td>6</td>
</tr>
<tr>
<td>4 Concurrency and Data Sharing</td>
<td>6</td>
</tr>
<tr>
<td>4.1 Data Sharing in Modules</td>
<td>6</td>
</tr>
<tr>
<td>4.2 Data Sharing in Subprograms</td>
<td>8</td>
</tr>
<tr>
<td>5 Non-determinacy</td>
<td>9</td>
</tr>
<tr>
<td>5.1 Parameter Passing</td>
<td>9</td>
</tr>
<tr>
<td>5.2 Function Definition</td>
<td>10</td>
</tr>
<tr>
<td>5.3 Select Statement</td>
<td>11</td>
</tr>
<tr>
<td>6 Exceptions</td>
<td>11</td>
</tr>
<tr>
<td>6.1 Propagation of Exceptions</td>
<td>11</td>
</tr>
<tr>
<td>6.2 Exceptions and Parameter Passing</td>
<td>12</td>
</tr>
<tr>
<td>7 Generic Subprograms and Modules</td>
<td>13</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>14</td>
</tr>
</tbody>
</table>
Abstract

In this paper we examine Ada with regard to program verification and make certain suggestions towards writing potentially provable Ada programs. We attempt to isolate and discuss those features of Ada which are not susceptible to current verification techniques. From verifiability considerations, the most critical features in Ada appear to be those which deal with data sharing under concurrent processing, direct referencing of non-local variables, access variables, "approximate" data-types, and generic program units. The independence of program units along with well defined interfaces for interactions is presented as desirable not only from software engineering aspects but also from the formal proof considerations. However, the possibility of having a large number of variables, potentially sharable among concurrent processes, is likely to make the proofs of Ada programs unmanageable. It is asserted, however, that with a certain discipline on the programmer verifiable programs can be written in Ada.
1 Introduction

The desire to construct deductively provable software has prompted intense research interest into the techniques and implications of program verification in recent years. This research has profoundly influenced the area of programming language design, for instance in the design of languages such as Gypsy [GCH78] and Euclid [LGH78].

Not surprisingly, the Department of Defense in specifying criteria for the design of a new standard language included in those criteria directives applicable to questions of reliability and verifiability.

"The design should avoid error prone features, should aid the production of reliable programs, should facilitate automatic detection of programming errors, and should aid in proofs of program correctness." [DOD1]

Ada, the language which ultimately emerged in response to these criteria is a large complex language of considerable generality and power. Certain features of Ada do not "aid in proofs of program correctness," and in fact are either extremely difficult or impossible to verify. This paper is an attempt to isolate and discuss such features of Ada with respect to existing techniques of program verification.

The organization of the paper is as follows. The following section describes criteria under which this evaluation was carried out. In Sections 3 to 7 these criteria are applied to various areas of the Ada design: data types, non-local referencing, multitasking, exception handling, and generics. In some instances restrictions are proposed which--either by incorporation into the language definition or by imposition as a programmer discipline--may aid in writing verifiable Ada programs. The final section enumerates conclusions of this research.
2 Verifiability and Language Features

There are various reasons why features which are not desirable from a verification standpoint appear in Ada. The conditions under which Ada was designed mandated some of these constructs. Steelman 6G, for instance, required the inclusion of a GO/DO. Though, not inherently unacceptable [KwU77] it is somewhat anachronistic in the presence of a complete set of more "structured" control constructs. The breadth of the intended applications area made certain other constructs necessary, e.g., floating point operations. Finally, some constructs were included which seem desirable and useful but for which it is not yet clear how susceptible they will be to verification. This is either because of their novelty or recent advent as programming language features. Generic subprograms seem to fall into this category. While recognizing that there are fairly compelling reasons why such features appear in Ada we feel that it is useful to point them out so that potential users have a sense of the limitations of the language with respect to program proofs.

Certain criteria are obvious in evaluating language features for verifiability. Results of applying language features should be well-defined; aliasing is to be avoided; features should lend themselves to formal axiomatization. Such factors must be considered in scrutinizing any language, including Ada. However, having been designed with some consideration for verification issues, Ada addresses them in a much more careful and deliberate manner than, for instance, an earlier language for which we performed a similar analysis, HALS [YO79]. Individual constructs of Ada were sufficiently well thought out that they almost entirely avoid aliasing, for instance. This suggests that in addition to these issues an analysis of Ada from a verification standpoint can profitably consider more general questions such as the overall program design strategies encouraged by the language and the way these influence proofs of Ada programs. However, the isolation and analysis of such issues depends upon subjective factors—the proof strategies one prefers, programming style, the level of formality required, the automatic assistance foreseen, the verification tools available, etc. Different evaluators will consider different points significant. Two such issues which we consider significant are the modularization of proofs and the "privacy principle."
2.1 Modularization of Proofs

Verification is a complex process. To reduce this complexity, it is desirable to borrow a strategy from software engineering and modularize proofs as much as possible. The possibility of doing this is contingent upon the degree to which operational and data abstractions are utilized to obtain modularity in program design.

Ideally, program segments can be written which behave in a determinate fashion which is completely capturable by a functional input-output relation associated with the unit. This permits the proof of each as a stand-alone unit without regard to the calling environment. However, the enforcement of this ideal in a uniform fashion by the programming language would require stringent restrictions on the parameter passing mechanism, concurrency, global referencing, etc. It seems reasonable to allow, as Ada has done, various classes of units--functions, value-returning procedures, procedures--which conform to this ideal in varying degrees. The question remains whether the particular restrictions imposed byAda are sufficient to permit large programs to be factored into manageable subunits for verification purposes.

2.2 The Privacy Principle

Another area illustrative of the interaction of verification considerations and language features is the "privacy principle". This principle for program proving is abstracted from the proof methodologies of Gypsy [GCK79] and Alphard [WLF76]. As applied to Ada it can be stated as follows:

The proof of any program unit should only make use of information which is available to that unit via the usual visibility rules of Ada.

This rule clearly encourages factorization of proofs as discussed in the previous section. Moreover, it imports the notion of "information hiding" implemented in Ada by private specifications into the proof domain. This permits operational and data abstractions to be effective uniformly--at proof-time as well as at compilation and run-time.

However, attempting to apply this privacy principle to Ada leads to the following questions. What sort of specification language for Ada will accommodate this requirement? Will the
visibility rules of the language carry over to the proof domain in a way which is sufficient to permit enforcement of the privacy principle? Do other features of the language such as the block structuring and global referencing make this principle impractical for Ada?

Issues such as the modularization of proofs and the privacy principle require more than a construct by construct evaluation of the language with respect to various criteria of verifiability. They require experience in trying to verify Ada programs of a reasonable size. Hence our comments here are intended more as suggestions for continuing research than as documentation of a completed analysis of Ada with respect to verifiability.

3 Data Types in Ada

The collection of issues which surround the area of data types within a language profoundly influences the verifiability of the language. Ada is a strongly typed language—a plus for verification. Moreover, the typing mechanism is fairly carefully defined, allowing all types to be known at compile time. However, a few issues remain which are noteworthy from a verification standpoint.

3.1 The Approximate Types

Ada has facilities for defining both fixed point and floating point data types with associated error bounds. At present, techniques are not available for verifying operations on such data types. This stems from a characteristic of operations on such types which is expressed in Ada by calling them "approximate" types. Each instance of an operation on an approximate type introduces an error term the magnitude of which depends upon the type of operation, the sizes of the operands, the implementation, etc. Sophisticated numerical techniques and error analyses are needed to make valid assertions about the results. The ability in Ada to specify error bounds does not substantially alleviate this difficulty.

The axioms which apply to arithmetic on real numbers do not in general apply to these operations. In this regard, arithmetic on approximate values is unlike integer arithmetic for which there exists a "ready-made" axiomatization which yields exact results except at very large positive and negative values. A very complex set of axioms probably would be needed for approximate operations. At present no convenient axiomatization
is known. The data dependent difficulties which lead to accumulating errors, catastrophic cancellation, etc., cannot be easily dealt with in axioms.

3.2 Access Types

Access types in ADA model fairly closely the pointers of Pascal. Ada does make certain restrictions on access variables [RAT79, 6.2.3] to avoid dangling pointers and aliasing of static variables. Access variables may designate only unnamed dynamically created objects. This limits the possibilities for aliasing, the verifier being assured that no named data item is also accessible via pointers. However, access types in Ada, as in Pascal, are fairly unregulated with regard to pointer movement, though proof strategies for Pascal pointers are in fact available [LUC79]. Ada, just as Pascal, permits movement of data objects by assigning pointers rather than by copying. This overloading of the assignment operator for access types permits easy construction of any number of aliases for a data object. The difficulties which such aliases present for verification are well known. A possible solution to this problem would be the definition of a new operator for pointer movement which does not leave the right hand side intact. This approach, along with appropriate restrictions on parameter passing, would make aliasing in Ada much easier to deal with.

4 Concurrency and Data Sharing

Sharing of data due to non-local referencing can make verification of programs difficult by permitting the aliasing of variables and a tighter coupling between various program units which share such variables. In proving program units which share data one must consider the potential interactions with all the other possibly concurrent processes which update shared variables.

4.1 Data Sharing in Modules

Proof of a module involves proving the routines defined within the body of the module. The routines must be proved individually as well as a collection considering their interactions among themselves and with other concurrent tasks. In Ada such routines can directly access non-local variables in accordance with the scope rules of the language. Most of the major obstacles in proving modules arise because of the concurrency of operations on shared variables. Concurrency can be exploited in two distinct domains, inter-task concurrency with
two or more tasks executing and possibly sharing global data, and intra-module concurrency with concurrent (or reentrant) calls to routines in a module body. These routines share the data declared in the module's body, and can also access some data non-local to the module.

One promising technique for proving a set of concurrent processes which share global data is that of Owicki and Gries [OWI76]. According to this method every process is first proved as a sequential program, and then a non-interference proof is constructed which demonstrates that the concurrent execution of the two processes does not violate their independent proofs. Decoupling, as far as possible, of the proof of one module from the proof of another is essential to make this verification scheme manageable. Therefore, we propose that the programmer impose upon himself the following discipline: any data declared outside a module should not be accessed directly by the routines defined in the body of the module. This implies that all the interactions between a module and the world external to it must be through the routines and the entries in the visible part of the module.

From the proof point of view, the "entry" mechanism of interaction between tasks appears to be more attractive than the use of shared variables. Input and output assertions can be associated with each entry of a task just as with subprograms. If the parameters of the entry satisfy the input assertion then after the execution of the accept statement the output assertion is expected to hold. However, unlike the subprogram calls the code executed for calls to a given entry can be different depending on the accept statement making rendezvous with the entry call. Therefore, if a pair of input-output assertions is associated with an entry, then it must be satisfied by all the accept statements for that entry.

Proofs of tasks in the absence of shared variables are like proofs of network of processes. Inductive proofs methods such as those used in proving concurrent programs in Gypsy [GCH79] and axiomatic proof techniques [CHA79] can be used in proving tasks which don't share variables.

Reentrancy and concurrency of subprograms declared in the visible part of modules need some special attention. These subprograms reference variables declared in the module and can be potentially concurrent when called by different concurrent tasks. Again from considerations of relative independence of proofs, any synchronization desired among the reentrant and possibly concurrent subprograms must be implemented within the module.
itself.

The input-output assertions associated with the entries and routines declared in the visible part of a module should only refer to the corresponding parameters and possibly some abstract properties of the module. This is necessary to comply with the privacy principle discussed earlier. Thus, the proofs of these entries and routines must be constructed at two levels: an abstract level and a concrete level. The proof at the concrete level refers to the data structures private to the module, whereas the abstract level proof does not refer to the private data of the module and must be derived from the concrete proof making use of the abstract properties of the module. This may require a definition of the relationship between these abstract properties and the structure of the module.

4.2 Data Sharing in Subprograms

Proofs of subprograms which access data shared by other concurrent processes can become quite tedious. Based on proof modularity considerations it is desirable not to allow direct accessing of shared variables inside programs. Subprograms provide a mechanism for textual as well as procedural or functional abstraction. One would like this abstraction facility to be preserved in the proofs as well. Unfortunately, direct accessing of non-local variables and passing shared variables as parameters (when parameter passing is by reference) to routines are likely to destroy this abstraction facility in proofs. Purely local analysis will not suffice in constructing the proof for such a subprogram.

As noted above, the demonstration of non-interference of proofs is essential in proving concurrent programs using the Dijkstra-Gries approach. In proving any program one must have a knowledge of which other concurrent processes share data with this process. A procedure accessing shared variables must be proven by showing non-interference of its proof with other possibly concurrent processes, which other processes execute concurrently with a procedure is dependent on the points in the program where the procedure is being called. This is true even if access to shared data is via the parameter list of the procedure. This means that a procedure which accesses shared variables cannot be proved without looking at the context in which it is called.

To insure that proofs of subprograms be possible as independent units irrespective of the context in which they are called, the programmer should adopt the following discipline: no
non-local referencing of shared data should be permitted inside subprograms. Regarding passing shared variables as parameters, the programmer must devise some synchronization mechanism to guarantee exclusive access to these objects before passing them as parameters to any routine. This assures modularity and enables the routine to be proved independently of the context in which it is called. One exception to this principle is necessary. Subprograms declared in the visible part of modules have to have access to the data declared inside the body of the module, otherwise their utility will be totally lost. Therefore it is impossible to maintain a uniform restriction on all referencing of shared data by subprograms. However, such a restriction is valuable for program units other than those declared inside the visible parts of modules.

5 Non-determinacy

A few features of Ada raise the possibility of non-determinacy in computational results. With some of these, the select statement for instance, the non-determinacy is intentional and complicates verification only insofar as it creates additional control paths. In other cases the non-determinacy results because the language definition does not specify adequately the results of certain operations. It is this unintended non-determinacy that we are concerned with in this section.

5.1 Parameter Passing

A rather unfortunate design quirk of Ada presents some difficulty for verification. The designers of Ada have refused to commit themselves to a particular mode of parameter transmission for out and in-out parameters. The language does not specify whether such parameters should be passed by reference or by copy. This potentially causes non-determinacy if in parameters of a sub-program are passed as out or in-out parameters to a procedure called within. Consider the following Ada procedure declarations:
Procedure Increment (x: in out t) is 
begin 
  x := x + 1;
end Increment;

Procedure P (y: in t) is 
begin 
  Increment (y);
end P;

Upon completion of the execution of P, it is undetermined whether 
or not the actual parameter corresponding to formal parameter y 
is altered in the calling environment. If in out parameters are 
implemented by copy it would not be, if reference it would. The 
following simple rule would eliminate this difficulty: a routine 
may not pass its parameters as out or in out parameters to 
other subprograms.

If parameters are passed by copying, non-determinacy could 
be exhibited in the case of parameters passed to concurrently 
executing procedures. The results in such a case would be 
dependent upon the execution speeds of the procedures involved. 
This introduces the complexity of concurrency into a domain that 
should be purely sequential. Other potential effects of this 
copy-reference ambiguity arise in connection with exceptions and 
are discussed in section 6.2.

5.2 Function Definition

Ideally, functions should describe an invariant in/out-output 
relation on the parameters. However, due to non-local 
referencing and the fact that functions may query system defined 
attribute functions (such as ACTIVE, PRIORITY, CLOCK, COUNT, etc.) 
which may be concurrently altered, the results of functions in Ada may be non-determinate or time-dependent. Such 
functions may not be provable independently of the global 
environment.
5.3 Select Statement

When an open alternative of a select statement is selected for execution, the condition associated with its WHEN clause might not remain true at the instant of execution of that alternative. This is possible because the WHEN clause might reference shared variables or call attribute functions which are concurrently updated [RAT79, 11.4.10]. This is not easily dealt with by existing verification techniques for such constructs which assume that for a guarded command \( S \rightarrow S' \), that \( S \) holds when \( S' \) is executed. The interaction of this feature with the other areas of the language, particularly the exception handling necessary to recover from such a situation, bears further study.

6 Exceptions

The possibility of exceptions occurring within a program adds considerable complexity to the verification process. Exceptions create additional, often implicit control paths which must be dealt with by the verifier. To find all such paths in general requires that for each statement one can enumerate all exceptions which might be raised by execution of that statement or to prove that none can be. For these reasons only a few attempts have been made to carry out formal verification in the presence of exceptions, the most notable being Gypsy [GCH78].

6.1 Propagation of Exceptions

Ada incorporates an elaborate exception handling capability akin to that of PL/I. Exceptions in Ada are viewed as anything deemed by the system or by the programmer to be an abnormal condition—they are not necessarily errors. When an exception is raised the containing routine is terminated and a handler is invoked. If no handler is locally defined, the exception is propagated outward through enclosing dynamic environments until an appropriate handler is found. Propagation of exceptions follows the dynamic chain, a fact that makes it both inconsistent with the static scoping of the rest of the language and inherently more complicated to verify than if it were done lexically.

It is important to be able to determine from a lexical scan of the routine text the potential exception conditions. The verifier must construct roofs of the various control paths including those involving the raising of exceptions. For locally handled exceptions these paths involve jumps to the handler; for unhandled exceptions the paths terminate abruptly with respect to
the local environment. However, a relatively high level routine may potentially be the recipient of a multitude of exceptions propagating from lower level environments. This not only complicates the verification process by multiplying control possibilities but also makes analysis on a purely local basis impossible.

Moreover, the information hiding aspect of procedural abstractions may be violated. Many propagated exceptions will have little relevance on the higher level—will be specific to lower level routines or anonymous. In the worst case, privacy may be compromised. An 'INDEX_ERROR', for instance, propagating from a stack manipulation package gives clues to the implementation which might otherwise remain hidden.

These considerations suggest that an alternative approach explicitly rejected by the Ada designers (RA77) be reconsidered. That is, unhandled exceptions could be converted to a standard ROUTINE_EXCEPTION before propagation. Explicit propagation, if desired, could be accomplished by re-raising the exception within the handler. This solution would reduce the verification complexity by reducing the number of potential control paths. Proof modularity would be enhanced since explicit errors could propagate at most one level. Finally, the possibility of inadvertently disclosing information would be removed. A programmer can easily implement this solution by including within each routine a handler with the "others" option which raises a ROUTINE_EXCEPTION which he has globally defined.

6.2 Exceptions and Parameter Passing

As noted in Section 5.1, Ada does not make a commitment to a particular mechanism for the passing of out and in-out parameters. The implementer is free to choose either a reference or copy mode of parameter transmission. The Ada Preliminary Reference Manual asserts:

In the absence of aliasing the effect of a subprogram call is the same whether or not copying is used for parameter passing, unless the subprogram execution is abnormally terminated. [REF74, n-3]

However, given that the exception mechanism is a significant part of the language this scarcely seems adequate. One exception, after all, can cause abnormal termination of several enclosing scopes. For any of them with out or in-out parameters, the
effects on their environments may be undefined. The following procedure definition illustrates this point:

Procedure P3 (x: in out integer;
    y: in integer) is
begin
    for i in 1 .. 1000 loop
        x := x + 1;
        y := y * y;  -- potential overflow
    end loop;
end P3;

The assignment to $y$ will almost certainly cause an overflow which will be propagated to the calling environment. However, when the exception occurs, the value of the actual parameter corresponding to formal parameter $x$ may be unchanged or may have increased by some indeterminate amount, depending upon the parameter passing mechanism. Verification of the calling environment will be hampered because the values of the variables will be uncertain. From the programmer's point of view it will be difficult to recover since it is impossible to know the effects of the partially completed routine on its environment.

A recent paper by Luckham and Polak [LuCa79] describes a technique, which is very similar to the methods of Gypsy [GCh76], for the specification and verification of exceptions in Ada. It is noteworthy, however, that the Ada execution environment is assumed in their work specifies that parameter passing is by reference and that exceptions are not propagated beyond their scopes.

7 Generic Subprograms and Modules

The generic definition facility provides Ada programmers a mechanism for translation level parameterization of subprograms and modules. This enables the programmer to write a generic package for stack manipulation, for instance, with the type of stack elements provided as a generic parameter. Little research has been done in the direction of proving such generic program units, the most significant being the verification work on Alphard "forms" [WiP76]. The verifiability goal for such units is "generic verification"—verifying the code once, in generic form. Currently the only approach available is to prove generic subprograms for each instantiation. Though adequate, this
approach is less than satisfying.

Generic verification in Ada is hampered by the generality of the generics facility. When a type, for instance, is given as a generic parameter additional "operator parameters" may be specified to operate on objects of that type within the body of the generic routine. At instantiation, the actual parameters supplied need merely match the parametric attributes of the corresponding formals. However, in constructing the proof of a subprogram one needs additional information, namely the entry and exit specifications for all operators applied to objects of the type within the body of the generic routine. These are not available to the verifier because the actual operators to be used are only determined at the point of parameter instantiation. Required is a generic specification capability and a means within Ada of syntactically delimiting the class of operators which can instantiate these operator parameters. Ongoing research in this area [YD80] indicates that generic verification might be possible for suitably restricted Ada generic units if these requirements could be met.

8 Conclusions

We have noted a significant number of features of Ada which do not lend themselves well to existing verification techniques. The context in which Ada was created explains the presence of many of these. Many more are due to the current state of available verification techniques. With continuing research into techniques for verifying concurrent programs, generics, modules, approximate types, etc., more of these features will become susceptible to verification. With the interest which Ada will doubtless generate much of this research will likely be directed to the specific goal of increasing the verifiable portion of the language.

Most of our evaluation has been based on criteria developed in evaluating the MAU/S language for verifiability [Y079] and from comparisons with Gypsy [CM78]. However, these languages are sufficiently different from Ada that these criteria may not be entirely adequate. Future experience in trying to design verification tools for the language and in trying to verify Ada programs will present a more complete picture.
Acknowledgements:

This work benefited decidedly from discussions with Dr. J. Misra and Richard M. Cohen. Support for this project was provided by The Defense Advanced Research Projects Agency under contract number N00039-78-G0080.
REFERENCES


