EVALUATION OF VERIFIABILITY IN HAL/S

William D. Young
Anand R. Tripathi
Donald I. Good
James C. Browne

ICSCA-HAL-1
(ICSCA-CMP-17) November, 1979

Institute for Computing Science and Computer Applications
The University of Texas at Austin

*This work was supported by NASA grant NSG-1479
TABLE OF CONTENTS

1.0 INTRODUCTION .................................................. 1

2.0 FEATURES FOR AEROSPACE SOFTWARE AND VERIFIABILITY ...... 1

3.0 CRITERIA FOR LANGUAGE EVALUATION .......................... 3

   3.1 Simplicity .................................................... 3

       3.1.1 Size ..................................................... 3
       3.1.2 Generality And Extensibility .......................... 4
       3.1.3 Coherence ............................................... 4
       3.1.4 Precision ............................................... 4

   3.2 Program And Data Structuring ................................ 4

       3.2.1 Abstraction And Extensibility ...................... 4
       3.2.2 Independence And Access Control .................... 4

   3.3 Security And Verifiability .................................. 5

       3.3.1 Aliasing ............................................... 5
       3.3.2 Axiomatizability ....................................... 6
       3.3.3 Simplicity Of Verification ........................... 6

4.0 VERIFIABILITY AND HAL/S ........................................ 6

   4.1 Aliasing Of Variables In HAL/S .............................. 7

   4.2 Axiomatizability ............................................ 8

   4.3 Non-determinacy In HAL/S .................................... 9

   4.4 Simplicity Of Verification .................................. 12

5.0 CONCLUSIONS .................................................... 12

REFERENCES ......................................................... 14
EVALUATION OF VERIFIABILITY IN HAL/S

1.0 INTRODUCTION

The HAL/S [1,2] programming language was designed to fill the software needs of the NASA space shuttle project. The breadth of this work mandated a language broadly based enough to meet the requirements of a wide range of aerospace applications. Some of these requirements—expressiveness and power in the language, ease of programming, system programming capabilities, orthogonality—are well served by HAL/S. For others, the situation is less satisfactory. In particular, the ability to write verifiable programs, a characteristic which is highly desirable in aerospace applications, is lacking since many of the features of HAL/S do not lend themselves to existing verification techniques.

This circumstance is not due primarily to carelessness on the part of the language designers. Rather it results because the special requirements of software for aerospace applications make conflicting demands on the designers of a programming language for this area. In the following section we discuss these conflicting demands and the burden they place on the programmer concerned with writing verifiable code. Section 3 describes the techniques of language evaluation and the means in which language features are evaluated for verifiability. These techniques are applied in section 4 to various features of HAL/S to identify specific areas in which the language fails with respect to verifiability. Finally we draw conclusions for the design of programming languages for aerospace applications and describe ongoing work to identify a verifiable subset of HAL/S.

The evaluation of the various features of HAL/S is based on work carried out at the University of Texas to identify a verifiable subset of the language. The criteria of evaluation, discussed in section 3 of this paper, were initially only vaguely formulated. We may therefore have been too harsh in our criticisms of some features and too lenient for others. The work is continuing and our goal is a subset which is both a valuable computational tool and a language in which reliable, well-structured programs can be easily constructed.

2.0 FEATURES FOR AEROSPACE SOFTWARE AND VERIFIABILITY

Any programming language designed for a specific applications area must be able to easily and naturally handle most of the computing needs for that area. For aerospace applications this requires a powerful, expressive medium adapted to scientific computation and the control functions of flight software. Desirable features include floating point capabilities, error handling and recovery, multiple data types with type conversion functions, and the ability to program for concurrent and real time applications. Also, since flight
software may be of considerable complexity the language should provide
a programming team the greatest assistance in constructing complex
software. Block structure, independent compilation, and interfaces
with subprograms written in other languages are desirable for this
purpose. HAL/S is a powerful tool for the aerospace programmer
precisely because it provides all of these features and more.

On the other hand, programs in any language are only worthwhile
if they correctly perform their intended function. This is doubly
true in aerospace applications where the failure of software may
derange human life and costly hardware. The ideal situation would
allow only programs of demonstrable reliability to be used in critical
applications.

A variety of techniques are available for establishing the
correctness of a program. Typically, programs are tested for some
subset of the possible data values. The selection of these test
values in HAL/S or in any language is related more to the complexity
of the problem than to specific features of the language. The amount
of confidence which can justifiably be placed in a tested program is
related to the degree to which the data domain is covered by the test
cases. Despite considerable research regarding the best way to
generate test data [7,8], absolute reliability is often the limiting
case as the size of the test domain approaches the size of the data
domain.

An alternate way to insure reliability advocated by Dijkstra [3]
and others is constructing all software via top-down structured
programming techniques. The reliability of a large structured program
derives from the reliability of small component modules with narrow,
well-defined interfaces between. Still the reliability is related to
the care with which the method is applied and the interpretation of
the subjective features of the methodology (how small are the modules?
how narrow are the interfaces?). Logical errors are still possible
because the only formulations of the desired goals of a piece of
software are in the programmer’s head and in the code he produces.
Structured programming does enhance the readability of programs and
help in isolating errors when they do occur.

The best approach currently available for software reliability
seems to be a combination of structured programming with program
verification. In this approach the program is proven to conform to
formal specifications for any value of the input domain. Assuming the
correctness of the formal specifications [6] and the soundness of the
proof rules, the verified program merits a high degree of confidence.

The following two difficulties arise when trying to apply the
techniques of program verification to a language such as HAL/S:

1. Proving the correctness of a program is often quite hard,
sometimes requiring more effort than designing the program to
begin with. This difficulty is compounded for large and
complex programs.

2. Many of the features which we noted as desirable for
aerospace software and which consequently are included in
HAL/S are difficult or impossible to verify.
The first point illustrates the need for structured programming to break a complex program into manageable units. Also needed are machine assisted program proving systems. Such systems are quite complex and as yet are limited in scope. One such system designed and implemented at the University of Texas provides machine assistance for proofs of programs written in the language GYP$SY$ (9). Though a general purpose language, GYP$SY$ was designed with verification in mind and contains few of the more troublesome features of HAL/S. The second difficulty is the subject of the following two sections.

3.0 CRITERIA FOR LANGUAGE EVALUATION

So far, few formal methods and tools are available for evaluation of programming languages, but the principles of language evaluation seem to be very closely related and in some sense isomorphic to the principles of programming language design - an area in which some work has been done in the past few years, particularly by Hoare (4) and Wirth (5). Our criteria for the evaluation of programming languages are derived largely from their work. The three major criteria are: simplicity; program and data structuring facilities; and security and verifiability. These are discussed briefly in the following section.

3.1 Simplicity

It is essential that the language be simple so that the programmer can have a good grasp of the features of the language in order to use them confidently. A major part of the verification process is eased if the programmer is well aware of what the program is doing and has a fair amount of confidence in it. In our opinion the simplicity of a language can be judged using the following four metrics: size, generality and extensibility, coherence, and precision.

3.1.1 Size -

Size of the language document and description is the most immediate measure of the simplicity of a programming language. A bigger document on language description naturally means more effort on the part of the programmer to learn and digest features of the language. Size of the description is closely related to the other three metrics and in particular to the generality and extensibility of the language. As discussed by Wirth (5), in order to keep a language simple the key is to include facilities which are simple to understand and are free from unexpected interactions rather than to try to minimize the number of basic features.
3.1.2 Generality And Extensibility -

The language should not attempt to provide separate features for every possible case which may arise. Rather the programmer should be provided with certain basic features with capabilities of easy and clear abstraction from these basic features to suit a particular problem. This would tend to keep the base language small without losing any generality of application.

3.1.3 Coherence -

Ideally the basic features of a language should be interaction free. In any case, side effects should be minimized and all exceptional behavior which might arise in combining language features should be clearly documented. Secondly, if a programming discipline is imposed by the language it should be uniform and unambiguous. Exceptional cases should be very much the exception in a good programming language.

3.1.4 Precision -

The semantics of the language construct should be precisely defined. For a truly simple language the definitions of the semantics can and should be concise without losing precision.

3.2 Program And Data Structuring

3.2.1 Abstraction And Extensibility -

Besides providing basic features which are easy to understand, a good language should have mechanisms for constructing new functional and structural abstractions. By structural abstraction we mean creating new data structures using the basic data structures provided by the language or using some previously defined data structures as building-blocks. Similarly, by functional abstraction we mean definition of new operations on the existing structure or newly constructed structure, in terms of the operations already defined on the building-blocks. For defining new abstract data types it is desirable to have capabilities to fuse these two kinds of abstractions in a coherent manner.

3.2.2 Independence And Access Control -

A key requirement on the language, for the realization of new abstractions, is the existence of facilities for access control. The language should permit decomposition of the problem into well-defined, independent domains of data along with functions (operations) associated with them. These independent domains interact according to some well-defined protocols. It is necessary that the language
provide access control mechanisms to restrict inadvertent interactions between different modules in the program. Such features in the language make the design process amenable to decomposing a problem in a top-down fashion, and an efficient realization of the design in a bottom-up manner.

3.3 Security And Verifiability

Simplicity of the language is most essential for an informal judgment of the reliability of the programs. In looking at the verifiability of the language we would like to evaluate how easily formal methods for proving correctness of the programs would be applied to programs written in the language. In order to make this evaluation it is necessary to examine the applicability of these methods in the context of the following aspects of the language: aliasing of variables, axiomatizability of the constructs and simplicity of verification.

3.3.1 Aliasing

In program proof methods, assertions are made about the state of computation at certain points in the program. These assertions use names for referencing the variables involved. It is necessary that variables be referenced using only one name, i.e., there should be a unique path to reference a variable. This ensures that an assertion, using one name for a variable, is not invalidated by changing the value of that variable using some different name. There are several possible ways in which aliasing may appear in programs. One is the use of pointers to objects which have been assigned names in the declaration of the program. A solution adopted by some languages such as Pascal is to restrict the use of pointers to only dynamically created objects. A second possible way aliasing may be introduced into programs is by passing global variables as parameters to a procedure which changes those global variables in its body. As Wirth [5] points out, this kind of aliasing arises from projecting the roles of textual abbreviation and of parametric program decomposition onto the same facility. For example:

```plaintext
Declare a integer;
S: procedure assign (x);
    Declare x integer;
    a = 2;
    x = 1;
    {Assert (x=1) and (a=2)}
close S;
Call S assign (a);
```
The assertion ",(x=1) and (a=2)" does not hold when a is substituted for x as an actual parameter.

3.3.2 Axiomatizability

Axiomatizability of a language feature refers to the ability to write a collection of rules which completely and unambiguously describes the results of applying that feature to any data item. Such descriptions are essential for automatic verification because the verification system manipulates symbolic quantities which represent arbitrary input values. The results must be comprehensible in terms of general principles which are independent of actual input values. One technique for assuring axiomatizability is to insist that language constructs conform as closely as possible to their mathematical counterparts, for which axiomatizations are readily available. It is not possible in general to conform them entirely. Integer addition in HAL/S, for instance, is axiomatizable from a verification point of view even though the associative law of addition may not hold for values near the maximum integer representable for a given machine.

3.3.3 Simplicity Of Verification

In many cases even if it is possible to axiomatize a certain feature of the language and give a complete mathematical definition of its semantics, it may turn out to be a very tedious job to verify programs involving the construct. Language constructs should be easy to grasp and at the same time simple to axiomatize. Simplicity in axiomatization leads to ease in applying proof methods.

4.0 VERIFIABILITY AND HAL/S

In studying the HAL/S language, our primary goal was to examine the various language constructs from the point of view of verifiability and define a subset of the language in which to write programs which can be verified using formal proof methods. The following four subsections discuss some of the features which we found to be major drawbacks of HAL/S from a verification standpoint. They outline as well the restrictions we imposed to alleviate these difficulties in defining the verifiable subset. One of the major problems in the way of applying formal proof methods to programs arises if the language allows aliasing of variables in the program. The following subsection discusses this problem in HAL/S and presents the modifications to the language which we propose to eliminate aliasing. The second subsection mainly deals with those constructs which make axiomatization difficult or impossible. It also examines those constructs which do not conform to their mathematical counterparts in a satisfactory way, in particular HAL/S functions. Section 4.3 discusses the possibility of potentially troublesome non-determinacy
in HAL/S programs. Finally in section 4.4 we discuss some of the features of HAL/S which are very cumbersome from the point of view of formal verification.

4.1 Aliasing Of Variables In HAL/S

As discussed in section 1.3.1, aliasing can result if procedures or functions are allowed to access global variables. HAL/S scoping rules permit this kind of situation to arise. We eliminate this by not allowing procedures or functions to access global variables directly. They can access only those global variables passed to them as parameters.

HAL/S has two mechanisms for passing parameters, namely "input" parameters and "assign" parameters. "Input" parameters are read only types of parameters whereas the values of "assign" parameters can be changed by the called routine. Aliasing can take place if at the point of procedure call some variable is used as an actual parameter for more than one assign parameter. Consider the following example:

```
Declare a, b, c integer;

P: procedure (x) assign (y, z);
 declare x, y, z integer;
do
    y = 4;
    z = 5;
{assert (y=4) and (z=5)}
end;
close P;
do
    .
    Call P (a) assign (b,b);
    .
end;
```

When procedure P is called with 'b' as actual parameter for both formal parameters y and z, then the assertion which appears after the assignments to y and z inside the body of the procedure does not hold. Assignment to z may or may not change the value assigned to y in the previous statement, depending on the implementation. The question of greater concern is what value 'b' has after the return from the call to 'P'. We eliminate these difficulties in the verifiable subset by requiring that each actual parameter be unique.
An analogous restriction is that at most one dynamically selected component of an array may be passed as an assign parameter to a procedure. For example the call

Declare A array (100) integer;
Call P (a) assign (A(1), A(j));

is not allowed in the verifiable subset since A(i) and A(j) may in fact refer to the same element.

HAL/S permits using an element of an array as the control variable in "FOR" loops. In HAL/S, the following program is valid:

Declare A array (100) integer,
     i, j integer;

Do
  i = 2;
  j = 1;

  do for A(i) = 1 to 100;
    A(j) = 0;
    j = j+1;
  end;

end;

The program loop would never terminate because of the aliasing taking place when 'j' takes value 2 and A(2) is set to 0. Therefore in the verifiable subset of HAL/S we permit only simple integer variables to be used as control variables in loops.

The HAL/S name facility, which we are not discussing in detail here, has provisions for setting pointers to objects which have been declared using some name. The possibility of this kind of aliasing is removed from the subset by disallowing this feature completely.

4.2 Axiomatizability

Particularly troublesome for axiomatizability are scalar (floating point) operations. HAL/S provides a wide variety of these as well as operations on matrices and vectors which have scalar components. The representation of real numbers by scalar quantities introduces errors which tend to accumulate in complex ways when arithmetic operations are iterated a large number of times. Moreover, the amount of accumulated error is largely dependent on the particular data values. Consider the following section of HAL/S code:

A = 1;
Do For I = 1 to N;
   A = (A + B);
   A = (A - H);
End;
The assertion that \( A = 1 \) following this segment of code is proveable by automatic verification techniques if \( A \) and \( B \) are integer variables. However, any axiomatization which permits the same conclusion if \( A \) and \( B \) are scalars is mistaken. Depending upon the values of \( N \) and of \( M \), the resulting value of \( A \) could be very close to 1 or differ from 1 by a large amount. Theoretically though, scalar arithmetic is probably axiomatizable. To do so would require the introduction of an error term for every application of an operation (not just at the boundary values as with integer operations), but a verification system can operate most easily by applying its axioms to symbolic quantities which are assumed to be exact and representative of an entire domain of possible data values. Because of this divergence the solution we adopted in defining the verifiable subset was to drop the scalar (and consequently the vector and matrix) data types entirely.

Another trouble spot were HAL/S functions. The mathematical concept of function permits a complete understanding of the behavior of a function in terms of a time invariant relationship between inputs and the values returned. HAL/S functions differ from this in the following respects:

1. Functions are permitted side effects.

2. The input parameters of a function may be altered during the execution of the function by passing them as assign parameters to a procedure called within.

3. The value of a function may be time dependent if the function accesses global data or calls system defined routines such as CLOKTIME or DATE.

Thus axiomatizability is made difficult since the axioms cannot regard a function in the mathematical sense but must treat any function call as potentially altering the global environment and being dependent upon more than the values of the parameters. We addressed these problems in the verifiable subset by imposing the following restrictions.

1. Functions (and procedures) may access only that data passed explicitly as parameters. Since functions cannot make assignments to their input parameters this reduces the possibility of side effects for a function.

2. No input parameter may be passed as an assign parameter to a procedure called within. This prohibits the changing of parameter values during the execution of the function.

3. Functions may not call time dependent routines such as CLOKTIME or DATE.
4.3 Non-determinacy In HAL/S

Non-determinacy in results of HAL/S programs could arise either because of using certain constructs which lead to implementation dependent results or because of performing concurrent operations on shared objects.

An example of constructs which lead to implementation dependent results is the multiple assignment statement of HAL/S. Consider the case where A is an integer array of size 10 and 'i' is an integer variable. If 'i' has value 2, then after the execution of the statement

\[ A(i), \ i = 3; \]

it is not defined whether A(2) gets value 3 or A(3) is assigned value 3. We prohibit this kind of non-determinacy in the verifiable subset of the language by not allowing the use of the same variable in the main line and the subscript line on the left side of the assignment.

Another place where HAL/S leaves room for non-determinacy is the allowance for passing the same variable as both an 'input' and an 'assign' parameter in a procedure call. The results depend on implementation whether the input parameters are passed like 'copy' parameters in Pascal or like 'var' parameters.

A similar kind of problem arises if a procedure passes its input parameters as assign parameters to a procedure called inside its body. This is shown in the example below:

\begin{verbatim}
P: procedure (x,y) assign (a,b);
  declare x, y, a, b integer;
  do
    Call Q (x) assign (y);
  end;
  close P;

Q: procedure (alpha) assign (beta);
  declare alpha, beta integer;
  ...
...
\end{verbatim}

In this case, again depending on the implementation, the call to Q in the body of P could change the value of the actual parameter passed to P for the formal input parameter 'alpha'.

In the verifiable subset we propose the following rules to eliminate such cases:
1. The list of input and assign parameters at the point of call must be disjoint.

2. A procedure may not pass any of its input parameters to some other procedure as an assign parameter.

3. During a procedure, an actual parameter cannot appear for more than one formal assign parameter.

Execution of concurrent operations could be a potential source of non-determinacy. HAL/S provides facilities by which a process can gain exclusive access to a data object, but they don't seem to be adequate to ensure consistency of the results. If two or more processes are allowed to access an object concurrently, then in order to assert the state of the object at certain points during the execution of any one of the processes, it is necessary that that process be given exclusive access to the object at those points. This assures that the state of the object is not changed by any other process at that point and hence the assertion remains valid. Secondly, for the consistency of the results the order in which concurrent processes operate on the shared data must be ensured in some way. HAL/S provides lock groups to which shared data objects can be declared to belong. All accesses to the locked objects are done in "UPDATE" blocks which, when entered, lock all the lock group of any locked data items accessed within that update block. From the verification point of view it appears desirable to insist that all compool and global data items which are potentially sharable by concurrent processes belong to some lock group.

The restrictions imposed on scheduling of tasks in HAL/S must be mandated by the proof procedures available for concurrent programs. An attractive method of proving the correctness of concurrent programs seems to be the one proposed by (Wicki and Gries [10]). This method requires the knowledge of all possible concurrent processes in the system. Non-interference is shown between concurrent processes (tasks) and then each process is proved separately using the proof rules for sequential programs. To reduce the complexity of this task there is a need for a disciplined approach to task scheduling. We suggest that all scheduling be done at the top-most program level. This makes it easy to identify potentially concurrent tasks in the system. Once these are identified it is necessary to show their mutual non-interference. However, this can be shown at the level of update blocks rather than at the statement level.

HAL/S permits multiple concurrent invocations of reentrant procedures to share one copy of certain data item. This is dangerous from the point of view of verification and should not be allowed for the reasons cited above.

Also, many of the constructs concerning real time scheduling could lead to undefined results. In general, verification of programs using time dependent constructs is difficult because of the absence of any knowledge of the execution speeds of the processes.

In HAL/S, abnormal termination of a process by its parent process could lead to some state not easy to define in the program. In
particular, because the execution speeds of tasks may depend upon various real time contingencies it may be impossible to determine from the program text the values of compool and global data items which are updated by the terminated processes. For this reason, it is very difficult to have any knowledge about the state of the process at the point of termination. We propose the following rule to simplify verification:

A process may not be terminated if this process or any of its dependent processes modify any of the compool or global data items.

4.4 Simplicity Of Verification

Many of the features of HAL/S, though verifiable, pose difficulty in verification because of the very involved nature of their axiomatizations. For an example constructs like "SUBBIT" which allow the programmer to make conversions from one data type to other by accessing its bit representation are difficult to understand and to axiomatize. Their inclusion in the subset would make verification process somewhat tedious although not impossible.

The "REPLACE" statement in HAL/S is also undesirable from the point of view of simplicity of the verification. Use of such a statement generates a totally new program and therefore any verification is valid only in the context of a particular instance of the "REPLACE" statement.

As discussed earlier, verification of programs involving time dependent constructs has some inherent difficulties because the execution speeds of the processes are not known at the verification level. At present there are no well developed techniques available in this area. For example in the case of cyclic scheduling of a process until a certain time there is no prior knowledge of the number of times the process would be executed. In such a case it is difficult to make any assertions about the global state at the end of the last cycle.

5.0 CONCLUSIONS

Our experience in attempting to isolate criteria of evaluation for verifiability and apply them to the features of HAL/S has taught us the following.

1. The design of programming languages for aerospace applications must be a process of compromise between the goals of inclusiveness and expressiveness on one hand and verifiability on the other. Language constructs must be evaluated from both perspectives.

2. A clear understanding of the kinds of features which are verifiable will aid in future work in designing verification systems for HAL/S and other languages. We feel that such efforts will be applied to many languages as more powerful
program proving techniques become available.

3. Most users of the language are concerned with writing software which is correct, but not necessarily in the sense of formally verifiable. For them this work identifies those language features which are ill-suited to structured programming and which contribute to the writing of software which is difficult to read, modify, and understand.

Our efforts will be successful to the degree that users of HAL/S are able to write useful software which is demonstrably correct and that future language designers make verifiability a higher priority.

Acknowledgement: We are grateful to Dr. J. Mistra for discussions regarding HAL/S and the general problem of language evaluation.
Evaluation of Verifiability in HAL/S

References


