INTRODUCTION

"A proof of correctness guarantees that a program will run correctly every time it is executed." That statement is not necessarily true. Suppose, for sake of concreteness, that a valid proof of a Fortran program has been constructed. When this program was proved, it most likely was proved in isolation from the other software components which ultimately will be involved in actually making the program run. So, even though we have proved the Fortran program, one of these other components, or the system hardware, may malfunction causing the actual machine language program that is executed to produce an error. These comments are not an argument against proving programs at the Fortran level, but rather an indication of the eventual need for a completely proved computing system.

Although, certainly, the development of even a small scale system that is completely proved must be viewed as a long term goal, the Nucleus project described here is intended to be a small, initial step in that direction. This paper summarizes the project and evaluates the progress made so far.

THE NUCLEUS LANGUAGE

The Nucleus project revolves around the Nucleus programming language as described in Reference 1. Nucleus was designed with three major considerations in mind, program provability, processor provability, and expressability of non-trivial programs. These considerations were manifest in seven specific design goals.

1. Program Provable. Every program in the language must be provable by the inductive assertion method.

2. Structured Programs. The language should support the ideas of structured programming.

3. Provability of a Verification Condition Generator. It must be possible to prove the correctness of a verification condition generator for Nucleus.

4. Provability of a Compiler. It must be possible to prove the correctness of a Nucleus compiler.

5. Ease of Compilation. Nucleus programs must be easily compilable into almost any machine language.

6. Rigorous Language Definition. All aspects of Nucleus, both syntactic and semantic, must be defined rigorously.

7. Express Non-trivial Programs. Nucleus must be capable of expressing non-trivial programs such as its own verification condition generator and compiler.

The central problem in designing Nucleus was in balancing the first six design goals against the seventh, expressibility of non-trivial programs. Ultimately, the principle was followed of making the language just powerful enough so that the Nucleus verification condition generator and compiler could be written in Nucleus without a great deal of difficulty. The following is a very brief summary of the language.

Essentially, the structure of a Nucleus program is the declaration of global variables followed by the declaration of procedures. Procedures are recursive, but have neither parameters nor local variables, and there is no block concept. Every procedure has unrestricted access to every global variable, and those are the only variables to which the procedures have access. This simple, but very primitive method for accessing program variables was motivated by several considerations. First, the construction of verification conditions is somewhat simplified if the variable X always refers to the same data object whenever it appears in an inductive assertion, and simpler construction leads to a simpler proof of the verification condition generator. Second, certain problems in proving procedures with parameters had been pointed out by Hoare. Ultimately, the choice was made to avoid these problems rather than to solve them. Also, since there are no parameters and no local variables, the compiler would be considerably simpler, and hence, easier to prove.
This rather drastic decision about parameters and local variables now certainly seems to be overly restrictive. In particular, Hoare\(^2\) and Hoare and Wirth\(^3\) describe ways that can be used to construct verification conditions for the types of parameter passage found in Pascal, and also indicate ways of treating local variables. It is not yet clear, however, how much these features would complicate the proofs of the verification condition generator and compiler.

The types of data objects that can be declared in Nucleus are quite limited. There are three primitive data types INTEGER, BOOLEAN, and CHARACTER. A value of type CHARACTER is just a single character in the basic character set of the language, these values being used primarily in input/output operations. In addition to simple variables, there is only one other kind of data structure, singly subscripted arrays. An array is declared with a constant upper bound and has an assumed lower bound of zero. Nucleus was limited to these simple data objects because, essentially, these were the only objects for which program verification methods were known. Since that time some additional progress on more sophisticated data objects has been made by Burstall\(^4\) and Hoare.\(^5\) The restriction to only singly subscripted arrays was not dictated by verification methods, but was chosen purely for the sake of simplicity. This eventually proved to be a bad decision because this makes it quite awkward to use one of the favorite tools of systems programmers, the table. This problem became readily apparent when faced with writing the verification condition generator and compiler in Nucleus. Allowing at least doubly subscripted arrays would help greatly in writing useful Nucleus programs and would cost very little in terms of additional verification technique.

The statements available in Nucleus are much more sophisticated than the variable accessing method and data objects. The fundamental statement of the language is the assignment statement which is unique only with respect to implicit fault conditions in expression evaluation, divide and modulo by zero, integer overflow, and array subscript violation. If any of these conditions occur, the program halts.

The control statements in the language were influenced strongly by the ideas of structured programming. In addition to a recursive procedure call, statements of Nucleus include

\[
\text{IF (exp) THEN (stmtlist) FI} \\
\text{IF (exp) THEN (stmtlist) ELSE (stmtlist) FI} \\
\text{WHILE (exp) DO (stmtlist) ELIHW} \\
\text{CASE (exp) OF (alternativelist) ESAC} \\
\text{CASE (exp) OF (alternativelist) ELSE (stmtlist) ESAC}
\]

where an (alternativelist) is a (stmtlist) preceded by a list of integer labels. In addition to the preceding statements, Nucleus also has a restricted GO TO. The GO TO can jump to any point within a procedure, but it cannot cross a procedure boundary. Under this restriction, the GO TO does not cause any technical problems in constructing verification conditions.

The statements that were absolutely essential to have in Nucleus in order to write a realistic verification condition generator and compiler, but for which verification techniques were not known, were simple READ and WRITE statements. The form of the read statement is READ arrayname, and its function is to transfer the next input record into the array. Element zero of the array is set to indicate whether the record read was, or was not, an eof (end-of-file) record. If the record is not eof, the rest of the array, beginning at element one, is filled with the characters of the record. If the record is eof, the rest of the array is not changed. The WRITE statement behaves in a similar way.

The one statement that was put into Nucleus strictly for purposes of provability was the ASSERT, which is the means by which inductive assertions are embedded in the program. ASSERT statements have no effect on the operation of the program and are used only in proving its correctness. In appearance, ASSERT is like the ordinary Algol comment, except that the word ASSERT is used instead of COMMENT. This gives the ASSERT statement a very flexible range of expressive power, which allows one to state conveniently the complex properties of the verification condition generator and compiler that eventually have to be proved. This type of extremely loosely formed assertion is adequate for the construction of verification conditions because the construction process requires only a way of detecting program variables in assertions, and this can be done with the help of the variable declarations at the beginning of the program. However, if the verification conditions are to be proved automatically, a more precise assertion language must be used.

PROVABILITY OF NUCLEUS PROGRAMS

For the most part, features were included in Nucleus only if it was well-known how to construct verification conditions for them; the notable exceptions begin the input/output statements, procedures with side effects, and the fault conditions in expressions. This section gives examples of how these features can be handled. More detailed descriptions can be found in References 6 and 7.

First, consider the read statement in the READINTEGER procedure of the following program. The statement

\[
\text{CHARACTER ARRAY INLINE[80];} \\
\text{INTEGER X,Y,Z,INTVAL;} \\
\text{PROCEDURE ADDXANDY;} \\
\text{ASSERT IF P IN \([1,2]\) THEN K(P) IN \([1,10]\);} \\
\text{ASSERT IF P IN \([1,2]\) THEN NOT :REOF(P);} \\
\text{ASSERT IF P IN \([1,2]\) AND Q IN \([1,K(P)]\)} \\
\text{THEN :RDFL(P,Q) IS A DIGIT CHARACTER;}\]
ASSERT IF P IN [1,2] THEN :RDFL(P,K(P)+1) IS A BLANK;
ENTER READINTEGER;
X := INTVAL;
ENTER READINTEGER;
Y := INTVAL;
Z := X + Y;
ASSERT Z = DECINTVAL(:RDFL(1,[1,K(1)])) + DECINTVAL(:RDFL(2,[1,K(2)]));
EXIT;

PROCEDURE READINTEGER;
ASSERT NOT :REOF(:RDHD+1);
ASSERT IF Q IN [1,K(:RDHD+1)]
THEN :RDFL(:RDHD+1,Q) IS A DIGIT CHARACTER;
ASSERT :RDFL(:RDHD+1,K(:RDHD+1)+1) IS A BLANK;
READ INLINE;
I := 1;
INTVAL := 0;
ASSERT IF J IN [1,80] THEN INLINE[J] = :RDFL(:RDHD,J);
ASSERT IF Q IN [1,K(:RDHD)]
THEN INLINE[Q] IS A DIGIT CHARACTER;
ASSERT INLINE[K(:RDHD)+1] IS A BLANK;
ASSERT :RDHD = :RDHD.0 + 1;
ASSERT I IN [1,K(:RDHD)+1];
ASSERT INTVAL = DECINTVAL(:RDFL(:RDHD,[1,I-1]));
WHILE INLINE[1] NE " Do
   INTVAL := INTVAL * 10 + (INTEGER(INLINE[I]) - 27);
   I := I + 1;
E1HW;
ASSERT INTVAL = DECINTVAL(:RDFL(:RDHD,[1,K(:RDHD)]));
ASSERT :RDHD = :RDHD.0 + 1;
EXIT;

READ INLINE of procedure READINTEGER reads the
columns of the next record of the standard input file into the
array elements INLINE[1], . . . , INLINE[80]. It is assumed
that this next record is not an eof record, that there is a
continuous sequence of digit characters beginning in column
one of the record, and that this sequence is followed by a
blank. These assumptions are stated explicitly in terms of the
"system" variables, :RDHD, :RDFL, and :REOF, in the
three assertion statements at the beginning of the procedure.
:RDHD is the "read head" variable which always equals
the number of the last input record read and is advanced
automatically by the read statement. :RDFL(r,c) and
:REOF(r) are two functions that define the standard input
file. :REOF(r) is a boolean-valued function that is TRUE
if record r is an eof record and FALSE if not. For records
that are not eof, :RDFL(r,c) is the single character in column
c of record r. In the assertions, :RDFL(r,[a,b]) denotes the
sequence of characters in columns a through b, and K(r) is
an auxiliary function that is assumed, and used, strictly for
stating the properties to be proved. The effect of the READ­
INTEGER procedure is described by the two assertions at
its exit. Upon exit, INTVAL is equal to the decimal integer
value of the digit character string beginning in column one
of the last record read. Also, the value of :RDHD, upon
exit, is equal to its value upon entry, :RDHD.0, plus one.
The verification condition for the path involving the read
statement is

0.1 NOT :REOF(:RDHD+1)
0.2 IF Q IN [1,K(:RDHD+1)] THEN :RDFL(:RDHD +
1,Q) IS A DIGIT CHARACTER
0.3 :RDFL(:RDHD+1,K(:RDHD+1)) IS A BLANK
0a :REOF(:RDHD+1) IMPLIES INLINE.10 = "T
AND [1 LE $ LE 80 IMPLIES
INLINE.1[$] = INLINE.10]
0b NOT :REOF(:RDHD+1) IMPLIES INLINE.10 =
"F AND [1 LE $ LE 80 IMPLIES
INLINE.1([$] = :RDFL(:RDHD+1,$))
AND [81 LE $ LE 80 IMPLIES
INLINE.1[$] = INLINE.10]
0c :RDHD.1 = (:RDHD+1)
1 I.1 = 1
2 INTVAL.1 = 0
3.1 IF J IN [1,80]
   THEN INLINE.1[J] = :RDFL(:RDHD.1,J)
3.2 IF Q IN [1,K(:RDHD.1)] THEN INLINE.1[Q] IS A
   DIGIT CHARACTER
3.3 INLINE.1[K(:RDHD.1)+1] IS A BLANK
3.4 :RDHD.1 = :RDHD + 1
3.5 I.1 IN [1,K(:RDHD.1)+1]
3.6 INTVAL.1 = DECINTVAL(:RDFL(:RDHD.1,
   [1,I.1-1]))
This verification condition is read as an implication with the dashed line (---) corresponding to the implies operator. The antecedent is the conjunction of the lines above the operator, and the consequent the conjunction of the lines below it. The "1.1" that is appended to several of the variables is an "alteration counter." Each time a variable is altered, its alteration counter is increased by one in order to distinguish the different values it may attain as the path covered by the verification condition is traversed.

The verification condition term for the read statement is the conjunction of the lines labelled 0a, 0b, and 0c. Line 0a describes the effect of the read if the record read was an eof, and line 0b describes what happens if the record read is not an eof. Line 0c describes the automatic incrementing of :RDHD. In this particular verification condition, line 0.1 affirms the antecedent of line 0b, thus giving the part of the consequent of line 0b that is needed to prove the consequent lines of the entire verification condition.

The treatment of Nucleus procedures is illustrated by the single verification condition of procedure ADDXANDY which consists of two calls of procedure READINTEGER.

method of definition

Probably the most unique feature of Nucleus is the method used to define it. The syntax of Nucleus is defined by two separate transition networks, one defining the parser, and the other the scanner. Both networks use the extensions described by Woods which allow the recognition of context-sensitive languages. In addition to defining the syntax, the parsing network also defines a translation from Nucleus programs into code for an abstract machine. The operation of the abstract machine, then, is defined by axioms as suggested by Burstall. This method was a major contributing factor to the provability of the verification condition generator. A more complete description of the method than the one given below can be found in Reference 1.

The networks used to define the Nucleus parser and scanner follow closely the ideas of the "augmented transition network grammars" described by Woods for dealing with natural languages. The basic idea is that a set of programmable registers is associated with the network, and each transition arc in the network is allowed to have a test and a sequence of actions defined on the registers. The test, is considered to be part of the transition condition from one state to the next, and when a transition is made, any actions that are associated with the transition arc are performed. This allows the network to specify the context-sensitive features that exist in most actual programming languages. For example, frequently it is required that an identifier cannot be refer-
enced before it is declared and this type of constraint usually is handled in some ad hoc way. This constraint can be specified quite easily using the tests and actions of the augmented networks. For example, in recognizing the declarations of a Nucleus program, actions may set a register, say IDLIST, to be the list of all identifiers that are declared. Then, in the part of the network that recognizes Nucleus statements, each time an identifier is found, a test is made to see if it is in IDLIST. Thus, the augmented networks provide a uniform mechanism for defining completely the context-sensitive syntax of Nucleus, and ad hoc rules are not needed. This is extremely helpful in proving the correctness of the language recognition parts of the Nucleus generator and compiler.

In part, the semantics of Nucleus are defined by the parsing network which also specifies a translation from Nucleus programs into a set of well-formed sentences in the predicate calculus. For example, the sentences corresponding to the declarations and the READINTEGER procedure of the program given above are:

```
ARRAY(INLINE,80)
SIMPLE(X) SIMPLE(Y) SIMPLE(Z)
SIMPLE(INTEGRAL) SIMPLE(I)
READ(0,INLINE)
ASSIGN(1,I,1)
ASSIGN(2,INTEGRAL,0)
IF(3,INLINE[I] NE " 4,7
ASSIGN(4,INTEGRAL,(INTEGRAL+10) +
(INTEGRAL(INTEGRAL[I]-27)))
ASSIGN(5,1,1+1)
JUMPTO(6,3)
EXIT(7)
```

(Some of the details of these sentences have been omitted to make them more illuminating than intricate.) These sentences are what Burstall calls the "structural description" of the program. The semantic axioms, then, are defined in such a way that the execution of the program can be deduced, by the ordinary rules of logic, from the structural description. In the Nucleus definition, the structural description may be viewed as a program for an abstract machine, and the axioms as defining the interpreter for that machine.

The execution of a Nucleus program is defined as a function E from program steps (non-negative integers) into state vectors—that is, the execution is a sequence of state vectors E[0], E[1], ..., each state vector in the sequence is itself viewed as a function from names into values. Each state vector has in its domain the names of the declared variables as well as certain "system" variables such as :RDHD and a location counter called :LOC. From a function such as SIMPLE(X) in the program above, it can be inferred, from the axioms, that X is a name in the domain of every state vector in the execution. Since each state vector is a function, the value of X in, say, state vector four is E[4][X].

In addition to the "declarative" axioms that define the domain of the state vectors, there are also "evaluative" axioms that describe precisely how expressions are evaluated, including the fault conditions. A third set of "imperative" axioms define the transition from one state vector to the next. These imperative axioms define the effect in the abstract program of the functions such as READ, ASSIGN, IF, JUMPTO, and EXIT. These functions are interpreted as the abstract instructions of the program with the first argument of each function being regarded as the abstract address that contains the instruction. The imperative axioms also define precisely the conditions of program termination.

**IMPLEMENTATION**

It is clear that it is fairly straightforward to implement Nucleus from this type of language definition. Such an implementation in Pascal for the CDC 6600 is described in Josue. This implementation is a two pass compiler that corresponds very closely to the Nucleus definition. Pass I simulates the transition networks and translates the Nucleus program into its corresponding abstract program. This part of the compiler actually was used to help debug the translation defined by the networks. Pass II then compiles machine language from the abstract program. Although not done, it would also be a simple matter to write an interpreter for the abstract program which would correspond fairly directly to the semantic axioms.

This method of definition also leads to an interesting technique for debugging the semantics of the language. In principle, the entire execution of the program should be deducible from the abstract program and the axioms. Therefore, one way of debugging the axioms and the translation into the abstract program is to do the translation and then begin making deductions from the program based on the axioms. (Although it has not been done in any of the work on Nucleus so far, this could be done by an automatic theorem prover.) In using this deduction technique on Nucleus, semantic bugs often were detected in one of two ways. The most common bug was one of inconsistency in which an abstract program and the axioms led to a contradiction. This usually was caused by a multiple definition of some component of a state vector. The other common error was one of incompleteness. In this case some point in the execution would be reached from which it was not possible to deduce the next state vector.

The abstract program also can be used as the basis for driving a verification condition generator. Such a generator was implemented by Wang in Snobol 4 on the CDC 6600. This generator simply follows control paths through the abstract program, and constructs verification conditions using alteration counters much like the ones shown in the examples above.

An interesting extension was made to the Wang generator to permit simultaneous top down design and proof of programs. The generator was extended to allow a procedure to have entry and exit assertions, but to have its body specified as PENDING. Verification conditions are not generated for pending procedures, but they can be called. Since the entry and exit assertions of the pending procedure are present, the proper verification conditions can be constructed for their calls if it were known what program variables the pend-
ing procedure might alter. The generator takes a totally pessimistic view and assumes that the pending procedure potentially changes the value of every program variable. This means that the alteration counter of every program variable is increased across a call of a pending procedure. The effects of this quite drastic assumption can be nullified, however, by explicitly stating in the exit assertion of the pending procedure that whatever variables are of interest are not changed. With the pending feature, it is possible to do, in parallel, a top down design and proof of a program. The top level procedure can be written, and the second level ones specified as pending. At this point the top level procedure can be proved, and then we can proceed to expand and prove the second level procedures.

PROVABILITY OF THE NUCLEUS PROCESSORS

A proof of a Nucleus verification condition generator, written in Nucleus, is described by Ragland. This generator also follows the Nucleus definition very closely and consists of two parts. The first is a simulation of the transition network which does the syntactic analysis and translates Nucleus programs into abstract programs, and the second part constructs verification conditions from the abstract program. A proof of a Nucleus compiler has not yet been undertaken, but the first part of the Ragland generator could be used, in fact, as the first pass of a two pass compiler so that the only part that would remain to be proved would be the code generation phase.

The two parts of the generator are reflected in its proof. The transition network simulation part is proved using an equivalence proof, and the part that actually constructs verification conditions is proved using inductive assertions. The contribution of the transition network definition of Nucleus to the practical provability of the generator can not be overestimated. Since the networks give a procedural definition of the syntax of the language, and also of the translation into abstract programs, it was a relatively straightforward matter to show, on a segment by segment basis, that the simulation in the first part of the generator is equivalent to the defining network.

The proof of the second part of the generator is done by the conventional inductive assertion method. As a prelude to this proof, verification conditions were defined for all the possible semantic constructs in Nucleus, and it was proved, by hand, from the semantic axioms, that these verification conditions are valid. The inductive assertion proof of part two of the generator then shows that these valid verification conditions are written onto the standard output file.

This proof also is worthy of note because of the size of the generator. The entire generator consists of 203 Nucleus procedures, most of these occupying less than one page including assertions. Of these 203 procedures, 103 were proved by equivalence proofs, and 100 were proved using inductive assertions. One of the most encouraging aspects of this large proof was that, on the basis of subjective evaluation, 70 percent of the things that were proved were quite simple, and it appears that almost all of these could have been proved mechanically.

BOOTSTRAPPING CORRECTNESS

The ultimate purpose of Nucleus is to serve as the bottom level of a bootstrapping process for developing correct processors for other more sophisticated languages. The central idea is to develop a correct verification condition generator and compiler for Nucleus and use these to build processors for other languages. The key question, however, is how to construct the first correct Nucleus generator and compiler that are to serve as the basis for further bootstrapping. The claim is that an unproved generator, such as the one written by Wang, can be used properly to help build a proved generator, such as the one written by Ragland.

The Wang generator, extended with the PENDING feature, was used heavily in proving the Wang generator. The key point is that it is not important to the Ragland proof that the Wang generator work correctly on all Nucleus programs. What is important, is that it work correctly on one particular program, the Ragland generator. If the unproved generator does work correctly on that one program, then the proper verification conditions are constructed for the Ragland generator, and thus, its correctness depends strictly, and properly, on the validity of the proofs of the verification conditions. This clearly, then, raises the question of how one verifies that the unproved generator worked correctly on that one particular program. This was done, very carefully, by the standard debugging technique of comparing the program input and output.

This development of a proved generator from an unproved one leads to the observation of an interesting relation between conventional debugging by test cases and proofs of correctness with respect to developing program verifiers. First, note that the same technique above would apply if the Ragland generator had been used to assist in its own proof rather than the Wang generator, and also that the technique still applies if we use complete program verifiers rather than just verification condition generators. We now observe the following property: if a program verifier operates correctly just one time, using itself as input, and obtains a proof, then that verifier operates correctly for all input programs. Said more simply, if it works on one input, it works for all inputs. This relationship is discussed somewhat more fully in Good with respect to both verifiers and compilers.

CONCLUSION

With the exception of a proof of a Nucleus compiler, all of the design goals of Nucleus have been attained. Although this proof has not yet been attempted, the proof of the Ragland verification condition generator strongly indicates the feasibility of the compiler proof. In meeting the design goals of Nucleus new techniques have been developed for
proving programs with input/output statements, procedures with side effects, and fault conditions in expression evaluation. The main contributor to the attainment of the design goals was the language definition method consisting of transition networks and axioms. This method particularly contributed to the practical provability and easy implementation of the Nucleus processors.

The main thrust of the Nucleus project has revolved around the design of the Nucleus language so that the language could express non-trivial programs, we could prove those programs, and also process those programs with correct processors. Ultimately, the important programs to be written, and proved, in Nucleus are processors, such as verifiers and compliers, for other more sophisticated languages. When this goal is attained Nucleus will have served its purpose.

REFERENCES
