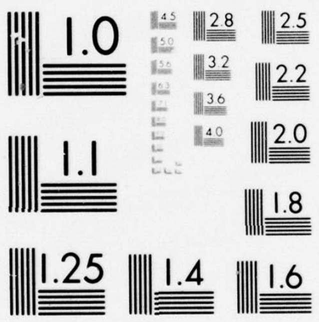
AD-A038 783 STANFORD RESEARCH INST MENLO PARK CALIF COMPUTER SCI--ETC F/G 9/2
THE DESIGN AND USE OF SPECIFICATION LANGUAGES.(U)
OCT 76 O ROUBINE N00123-76-C-0195
UNCLASSIFIED CSL-48 NL

1 OF 1
AD
A038783

END
DATE
FILMED
5-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

9/ (12)

Technical Report CSL - 48

October 1976

THE DESIGN AND USE OF SPECIFICATION LANGUAGES

By: OLIVIER ROUBINE

See Form 1473

Prepared for:

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

CONTRACT N00123-76-C-0195

W. LINWOOD SUTTON, Contract Monitor

STANFORD RESEARCH INSTITUTE Computer Sc. Lab.
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

Technical Report CSL - 48

October 1976

THE DESIGN AND USE OF SPECIFICATION LANGUAGES

By: OLIVIER ROUBINE

Prepared for:

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

CONTRACT N00123-76-C-0195

W. LINWOOD SUTTON, Contract Monitor

Approved by:

JACK GOLDBERG, *Director*
Computer Science Laboratory

EARLE D. JONES, *Associate Executive Director*
Information Science and Engineering Division

THE DESIGN AND USE OF SPECIFICATION LANGUAGES*

Olivier Roubine**

Computer Science Laboratory
Stanford Research Institute
Menlo Park, California 94025

ABSTRACT

The goal of proving the correctness of computer programs has been hindered by the difficulty of stating formally what is to be proven. The tools developed to achieve this latter purpose often receive the name of "specification languages", but the formalization and implementation of such languages has seldom been carried out, and very few of these languages have been described in the literature.

We present some design issues concerning specification languages, e.g., desirable properties of such languages, and their limitations, as well as some ideas on their use (and hence on the use of specifications in general) as an integral part of the design process leading to reliable software systems. The (possibly automated) verification of some properties of formal specifications is considered, and some conclusions are drawn concerning the help such an approach can bring in providing greater confidence in the final software product.

* This work was partially supported by the Naval Electronics Laboratory Center under Contract N00123-76-C-0195.

** The author is also associated with the Computer Science Department, University of California at Berkeley.

1. INTRODUCTION

Substantial efforts have been made in recent years to develop techniques for writing better programs more easily, and to prove that they behave correctly. In the first category are techniques described in particular in [Dahl], where the use of step-wise program composition (see also [Wirth]), hierarchical program structures, and data abstraction is strongly advocated. The formal verification of computer programs has also received much attention ([Elspas], [Manna 1]), and an even more challenging task is the automatic synthesis of correct programs ([Manna 2]). All these approaches to software development have at least one thing in common: the need to state what a program or an operation does, independently of its implementation. This is clearly a necessity in program verification, where the goal is to prove that a given program actually does what it is claimed to do, and in program synthesis, where the goal is to generate a program, knowing only what it is supposed to do -- and prove its correctness during the generation process. When using a hierarchical decomposition, the designer who writes one level of a program need not know how the "abstract operations" he provides are to be used, nor how the operations he himself uses are implemented. Since it is sufficient for him to know what these functions do, the need for specification techniques should become apparent. Similarly, program verification and synthesis can benefit from the use of data abstractions ([Liskov 1], [Guttag], [Flon], [Spitzen]), and the techniques used to specify them.

Our concern is how statements about the actions of programs are expressed, and we call the languages used for so doing _specification languages_. We restrict ourselves here to particular aspects of specifications. We shall not consider such issues as specifying the semantics of programming languages ([Tennent], [Wegner]), or specifying how programs operate: the specification of algorithms can be done in various ways including flowchart languages, a step-by-step description in some natural language, or the use of a high-level programming language. The reason we do not insist on algorithms is that some research in programming language design is aimed at providing very high-level languages that tend to be non-procedural (e.g., SETL - [Schwartz]) and where the specification of algorithms becomes of little concern to the user: to some extent, the specification language used in an automatic programming system can be viewed as a very high level programming language. It is interesting to notice how the trends for specifying algorithms have evolved in the past: with the primitive control structure provided by assembly languages and early languages such as FORTRAN, flowcharts enjoyed a certain popularity: their lack of readability was a drawback, and some designers preferred a description of an algorithm as a succession of steps, each step being described in a natural language and indicating what step should be performed next. However, the development of ALGOL 60, in particular, was somewhat troublesome in that the programming language used for implementing an algorithm had a more sophisticated control structure than the language used for describing the algorithm (the introduction of recursion was another torment for several programmers faced with the problem of specifying an algorithm using recursive function calls). Nowadays, the language PASCAL seems to be quite popular as a tool for specifying algorithms, but we can expect that the advent of a new programming language of higher level would make PASCAL obsolete as a tool for specifying algorithms. The issues we discuss here concern mostly the specification languages used to describe formally the effects of programs, functions and operations.

The remainder of this paper is divided into two main sections, followed by some concluding remarks. We first state some desirable properties that we think specification languages should exhibit, and present some of the conflicts between these goals. Following is a discussion of some issues that affect the way specifications can be used for software development, and how they foster or hinder the desirable characteristics of a specification language.

2. DESIRABLE FEATURES OF SPECIFICATION LANGUAGES

We distinguish three principal aspects of specification languages that should receive attention. Those are readability, ease of use and precision. These aspects can be compared with the criteria discussed in [Liskov 2], for the evaluation of specification techniques for data abstraction: in this reference, the issues are formality (which is very close to our notion of precision), constructibility (an element in the ease of use), comprehensibility (related to the readability of a language), minimality and wide range of applications (which are discussed in section 3 as influencing the design of a language), and extensibility of a specification method (which should not be confused with the extensibility of a language, discussed in 3.5).

2.1. Readability

The notion of readability is related to the need for specifications to be used by humans: if the only reason for a formal specification is to be used as input to an automatic program verifier, then the issue of readability is of less importance. However, when, in a large programming task, specifications are used as means of communicating information between the various teams involved in the project, then readability and understandability are fundamental. When some part of a software system is designed by someone and used by someone else, the specification should contain what the designer discloses about what his part of the system does: if what he says has little meaning for the one who uses his product, the specification will have missed part of its purpose. It is also worth noting that there is a need for specifications as a statement of purpose, similar to the formulation of a problem, for which the final software system is a solution. The specifications of what is to be done should ultimately coincide with the specifications of what the user can see of the final product. We think that techniques for defining a software system can be derived from the techniques used to describe a system, and we shall therefore concentrate on the latter.

Coming back to specifications of programs, consider the example of a program that sorts an array of integers A in non-descending order. We give below various suggestions on how to state that after the execution of the program, the array A is indeed sorted:

1) a comment, e.g.,

   (* The array A is sorted in non-descending order *)

2) a predicate of the form

   (ORDERED A)
   (example taken from [Boyer])

3) the assertion

   (LTQ (STRIP (TUPA A 1 N)))
   (from [Waldinger])

4) the mathematical formula

   $\forall\ i, j \in \{1, \ldots, n\}\ i<j \Rightarrow A(i) \leq A(j)$

In terms of readability, the first two statements are clearer, providing the terms "sorted", "non-descending", "ORDERED" are understood. We shall leave the appreciation of the third example (QA4 system) to the reader. As for the fourth suggestion, we think it is clear, concise and readable only beyond a certain mathematical training. The various forms shown above illustrate the concept of readability. They could be ranked according to this particular criterion, but we will also discuss them with respect to other considerations.

2.2. Ease of Use

The concept of readability evolves from a concern for the reader of the specifications. That of ease of use stems from some concern for the writer. The issues are: how easy is it for the program designer to express his conception of the program in terms of the specification language constructs? Are there any assertions he might want to write that would require a very complicated statement, or that would be impossible to write? (This last question defines the power of the language.) The specification language should be a tool to write precise assertions without diverting the designer from his main task by placing a heavy burden on him every time an assertion has to be made. It is to be noted that the originality of the syntax is less important: the language will have to be learnt anyway, and a cryptic but very powerful syntax (e.g., in a way similar to APL) may have its advantages in terms of writing specifications, although often at the cost of hindering the readability. Going back to our previous example, in the context of the Boyer-Moore theorem prover ([Boyer]), the assertion (ORDERED A) would also require that the user defines the term ORDERED as a LISP function, which is exactly the kind of effort the user wants to be spared from (not to mention the possibility of giving a wrong definition for the function ORDERED). Another gain in terms of ease of use is the ellipsis ({1,..., n}) notation used in the fourth example: if this particular notation is not provided, the formula must be rewritten:

$$\forall\ i,\ j\ (i \geq 1 \wedge i \leq n \wedge j \geq 1 \wedge j \leq n \wedge i < j) \Rightarrow A(i) \leq A(j)$$

2.3. Precision

If there is any hope of proving that a program meets its specifications with the help of an automated verifier, the semantics of a specification language must be formal enough to lend itself to the program verification process. Such a system generally works within some mathematical theory (e.g., predicate calculus, set theory), and specifications will have to be meaningful in the context of this theory. Specifications will have to be "interpreted" and understood. Another reason why the definition of a specification language should be extremely precise is that any imprecision in the language itself will lead to imprecise specifications, whose contribution to the reliability of the final software system will be very small, if not negative.

2.4. Tradeoffs and Conflicts Between the Goals

The enhancement of readability, ease of use, and precision is not always a very easy task: too much emphasis on one of the goals may compromise the other ones. We shall attempt to demonstrate some of the possible conflicts.

2.4.1. Readability vs. Ease of Use

If a specification is easy to write in a concise form, it will also often be easier to read. We mentioned earlier, however, that the writers of specifications tend to be much more familiar with the details of a specification language than the readers, because they need to understand all the constructs in order to compose specifications. The assertion 3 from section 2.1 is an example of an assertion that is concise, but may require a big effort from the reader. The example of an APL-like syntax can also show how the writer's convenience may diverge from the reader's.

2.4.2. Ease of Use vs. Precision

The ease of use of a specification language is generally enhanced by the inclusion of built-in constructs to express particular properties, e.g. the notion "ORDERED" applicable to a vector or a list. It should be clear that the more of these built-in constructs (which we call primitive constructs) are provided by a language, the harder will be the task of formalizing the language. In addition, a plethora of different constructs (i.e., more constructs than provided by the underlying mathematical formalism) will introduce some redundancy in the language itself, thus making it more prone to misuse. Redundancy also introduces possibility of various "specification styles", a theme that will be considered in section 3.

2.4.3. Readability vs. Precision

We will only mention here the case of the comment in the first part of our array example, which is probably very meaningful to anyone with a minimal knowledge of the English language, and on the other hand deficient in terms of precision: even if "non-descending" is a known term, "order" can be understood only in a particular context (and even we cannot be sure that the word refers to the natural order relation on integers, since if the designer chose to define a different order, the comment would still be applicable).

An example of an assertion that is extremely precise but not very readable is a variant of our mathematical formula. Assuming we are allowed neither the ellipsis notation, nor the multiple quantification, we might write:

$$\forall\, i\ (i \geq 1 \wedge i \leq n) \Rightarrow \forall j\ (j > i \wedge j \leq n) \Rightarrow A(i) \leq A(j)$$

but we can expect some readers to find it more readable than the expression

```
(LTQ (STRIP (TUPA A 1 N)))
```

3. ISSUES IN SPECIFICATION LANGUAGE DESIGN

3.1. Generality

A general purpose specification language should be applicable to a large class of problems, e.g., numerical analysis and system programming and specification of data structures and string processing. If the specifications are to be used for automatic verification, the particular verification system used will have to be able to reason in a large variety of domains, e.g., number theory, predicate calculus, set theory, integration theory, topology, not to mention a few areas that would have to be modeled, e.g., the concept of type. At the present time, it is not feasible to envision a practical verifier working in all these branches. Even if automated treatment of specifications is not envisioned, there is another reason which makes too much generality not only infeasible but also undesirable: the specification techniques that are suitable for a particular class of problems may reveal themselves awkward in another domain. For instance, in the specification of synchronization problems ([Griffiths]), global invariants may be of primary importance. On the other hand, when specifying the operations provided by an abstract machine ([Robinson 1]), such global assertions (e.g., POP(PUSH(x,s)) = s, where s represents a stack) may be redundant, or even superfluous. It is therefore our conclusion that a general purpose, yet practical, specification language would have to be too large to be manageable.

It seems therefore more reasonable to expect a specification language to be fairly small and limited to a particular class of problems. The language may then reflect a particular approach to these problems, as is described next.

3.2. Methodologies for Specifying Problems

One should not expect specifications to be written in an undisciplined manner: if various authors have advocated a "discipline of programming", this discipline should also be reflected in the specifications. If an operating system is to be designed using a hierarchical decomposition into modules ([Neumann], [Robinson 2]), then specifications will have to be written to describe modules. If one´s conception of data structures is in terms of sets ([Warren]) then one will have to write assertions about sets. An illustration of the diversity of the possible methodologies to handle a particular class of problems can be found in [Liskov 2], and we will not give an extensive description of the methods mentioned in this reference. We will note, however, that each methodology places some emphasis on certain "objects", e.g., graphs, labels and nodes ([Earley]), O- and V-functions ([Parnas]), operations and their properties ([Guttag], [Spitzen]), or sets ([Warren]). Our concern here is to see how the choice of a particular methodology affects the design of a specification language. A methodology reflects a vision of the world: when dealing with problems in Newton´s mechanics, one works with the notions of mass, force, length, time, and takes these notions as predefined. Similarly, the concepts existing in a methodology have to be incorporated in a specification language, such as, for instance processes ([Griffiths]), or graphs ([Cook]). The methodology will also rely on some domain of mathematics: for example, the algebraic methods of Guttag or Spitzen rely heavily on the notion of recursion and mathematical induction. On the other hand, the set-theoretical approach of Warren relies on set theory (this should come to no surprise). When using Floyd's method for verifying programs ([Floyd], [Hoare]), assertions (usually expressed in predicate calculus) must be used.

It is therefore our claim that a specification language can only be designed after the choice of a methodology for designing, writing or verifying programs has been made, and only in the context of a particular domain of applications.

Another criterion may affect the choice of a mathematical formalism within which specifications must be written, namely the availability of verifiers working in the particular formalism: for example, first-order predicate calculus may be more suitable, because, in spite of some efforts (e.g., [Darlington]), verification techniques using higher-order logic have not reached a practical point (see [Ernst]). Some approaches to verification may also place more emphasis on pure set theory, or recursive function theory (e.g., [Boyer]), and thus introduce a new constraint in the orientation of a specification language. Following [Kowalski], we would tend to consider predicate calculus as eminently suitable for expressing assertions in a generally natural form, "in that it derives from the normative study of human logic".

It should be clear that the formal methodology and the mathematical domain in the context of which specifications are to be written are key to the precision of the language: if some concepts introduced by the methodology are not defined in a sufficiently formal way, the precision of the language will suffer. Of particular importance is the definition of the classes of objects that are manipulated.

3.3. Objects in Specifications

Whereas a program operates on data and may manipulate data structures, a specification describes properties of a software system, and may therefore refer to objects manipulated by that system, but may also use some other objects in the process of describing a system: for example, when we specify that an array is to be sorted, we must have a representation in our specification of this array which is an object of the system being described; however, when we write

$$\forall\, i, j \in \{1, \ldots, n\}\ i<j \Rightarrow A(i) \leq A(j)$$

the objects i and j have no existence in the system, and exist only in the specifications. An analogy can be found in considering an architect's blueprint as a specification for a building: we will see thick lines supposed to represent walls (these are representations of system objects in the specification), and thinner lines possibly terminated by arrows and associated with a number, that denote particular dimensions. It is important for the builder to realize that the thinner line does not correspond to a part of the house he is building. Our point here is that the distinction between system objects and specification objects must appear clearly in the formalization of the specification language, lest the specification becomes meaningless (one does not build a wall out of measuring tapes). Similarly, the distinction between the objects manipulated in a system, the representation of these objects in specifications, and the objects that exist only in the specifications, must be made extremely clear. This distinction is also a prerequisite to the introduction of types in specification languages.

3.4. Types in Specification Languages

As described in Hoare's "Notes on Data Structuring" (in [Dahl], pp. 83-174), types are sometimes used by mathematicians to recognize different categories of objects (e.g., variables, functions, functionals, or values, sets, sets of sets). The notion of type was very early related to the representation of data inside a computer (e.g., FIXED(5,2)) very early in the history of programming languages. This relation is obviously of little interest in a specification language. More important is the fact that both in mathematics and in programming, the notion of type introduces the possibility of detecting expressions that are meaningless, regardless of the particular values of their components.

A similar feature might be desirable in specification languages for the reasons that make it appealing both to mathematicians and (to some extent) to programmers: types allow the programmer to make assumptions about the arguments to operators and functions and relieve him of the burden of having to make sense out of every possible value: types also allow a mathematician to apply some knowledge about properties of particular objects, when reasoning about these objects. For example, the knowledge that a value is a natural integer will permit the use of mathematical induction, whereas this would not work on reals.

An important distinction has been made between system objects and specification objects: if a specification language incorporates the notion of type, it should allow the user to distinguish between the types of system objects and the types of specification objects. In particular, if the system being specified manipulates numbers, and the specification language also provides for numbers, the specifications may lead to misunderstanding and confusion between the two categories of numbers: the general assertion

"for all specification_number i, there exists a unique system_number j such that i = j"

is likely to be left implicit. Note, however, that the assertion "i = j" would be meaningless in any strongly typed language, since i would be of type "specification_number" and j of type "system_number". This brings forth the issue of strong type checking: unless restrictions are imposed, it may be impossible to determine whether an object belongs to a particular set, and the question is how much constraint should the language impose on the user by forbidding the writing of expressions that do not abide by certain rules? For example, the union of a set of integers and a set of characters may be a perfectly reasonable set of objects (in particular for an input/output routine). It might be worthwhile to have an automatic checker process the specifications, detect some possible type conflicts and warn the user, but it might not be desirable to reject the specifications because of such conflicts. Too many constraints of this order will cause the users to express what they mean in specifications that are less clear, because the simplest way would cause a rejection of the specifications.

3.5. Extensibility

Types are a convenient way of stating structural properties of objects. In addition, the possibility of naming types tends to make specifications more readable by the use of relevant names (e.g., "stack").

The use of various kinds of definitional facilities (for types, declarations, expressions) can help make specifications more concise, more readable, and easier to write. An example of the gains to be achieved by such mechanisms is the expression "(ORDERED A)" given above, where ORDERED may have been defined at an earlier stage of the specification. Strong type checking requires definitional facilities of greater sophistication in order to achieve a given degree of flexibility, For instance, in a language that provides both sets and vectors of any type (e.g., SPECIAL - [Robinson 3]) it would be very convenient, on some occasions, to be able to talk about the set of all the coordinates of the vector, no matter what the base type is. Rather than providing a predefined set of constructs, much more flexibility could be gained from a minimal set of basic constructs (those identified by the methodology and the mathematical theories used to write specifications), together with suitable extension mechanisms. In this way, only a small language has to be formalized and understood in order to read, write and interpret specifications. On the other hand, if the language that is provided is too crude, the writer will have to be distracted from his main design effort in order to write a simple concept in the available syntax.

For instance, in set theory, the set inclusion can be easily defined in terms of set membership by:
S1 is a subset of S2 if and only if
$\forall x, x \in S1 \Rightarrow x \in S2$
Having to write such definitions in all his specifications is likely to be considered as a burden by the user. There is a subtle balance between providing too simple a language, in which each expression is fairly complicated, and a language that is so rich that there is a particular form for each possible expression, if only the user can remember which one. In addition, a language that is too uneconomical in its power of expression may be difficult to formalize.

3.6. Procedural Aspects of Specification Languages

We have excluded from our scope the languages for specifying algorithms. The specification of operations or data structures tends to rely on assertions that should be true in a particular state (e.g., before or after an operation is executed), or always true (e.g., axioms of the kind "POP(PUSH(s,x)) = s". The notion of sequencing, i.e., of an expression being evaluated before another, and that of transfer of control (e.g., procedure calls) are of little, if any, concern in specifications: when we write

$$\forall i, j \in \{1, \ldots, n\}\ i<j \Rightarrow A(i) \leq A(j)$$

we are not saying that this formula has to be true successively for i=1 and j=2, i=1 and j=3, etc. In fact, the procedure

```
function ORDERED (var A: array [1..n] of integer): boolean:
begin
     ORDERED := true:
     for i:=1 to N-1 do
          ORDERED := ORDERED ∧ (A[i] ≤ A[i+1]):
end:
```

introduces the unwanted notion of sequencing (note, however, that as it is written, the function will always be executed in a fixed number of steps, thus hiding the fact that all indices are not treated equally). Procedural specifications tend to be more complicated, forcing the user to devise an algorithm to evaluate a predicate.

There are some cases where some notion of sequencing has to be included in the specifications. For instance, in the language SAL ([Griffiths]), the concept of "trying" to wake up a process, and "then", in case of failure, try and wake up another one, is essential.

Note that this use of "then" is basically different from its use in the form "if P then Q else R" which can be translated very simply as
(P AND Q) OR ((NOT P) AND R)
One of the problems facing the designer of a specification language is that of devising a syntax that will remind a user of some procedural constructs. For instance, in SPECIAL ([Robinson 3]), boolean assertions may be separated by ':', "P: Q:" meaning that both P and Q are asserted. In this context, it sometimes comes as a surprise to some occasional users that

```
i = j:
j = k:
```

necessarily implies the relation "i = k". First, '=' does not connote assignment, and second, names generally do not stand for variables, in the programming sense.

It is interesting to notice how some people familiar with programming practices will try to transfer their procedural notions to non-procedural specification languages: for instance, another way of writing our predicate

$\forall i, j \in \{1, \ldots, n\} \; i<j \Rightarrow A(i) \leq A(j)$

could have been

$\forall i \in \{1, \ldots, n-1\} \; A(i) \leq A(i+1)$

This latter expression forces a reader (or a verifier) to use his full knowledge of the natural ordering of integers and the transitivity of the relation "$\leq$" in order to compare A(i) and A(j) for arbitrary i and j. In that respect, the style itself is more procedural in the second case than in the first.

The above example illustrates why procedural specifications are not desirable if they can be avoided: they tend to require a greater effort in order to be understood, or even some explanations, in the form of specifications (see the definition of ORDERED).

3.7. Some Syntactic Aspects of Specification Languages

A specification language is a desirable vehicle for communicating information between various people. Therefore one must be particularly careful about syntactic constructs that may have some connotation in the user's mind, different from its actual meaning, or on the contrary no connotation at all (e.g., STRIP).

One example is the choice of identifiers to translate mathematical symbols: if the language designer has a higher training in mathematics, he will in all likelihood attach to the symbols EPSILON and SIGMA the notion of set membership and arithmetic sum, respectively. The problem is to know to what extent these names will be as meaningful in the user's mind, and if a different choice (e.g., MEMBER_OF and SUM) would be clearer.

We have mentioned in the previous section how the '=' sign may be misleadingly interpreted as an assignment. One of the notions that may be unfortunately introduced in a language is that of sequencing, as in the case of the "if...then...else..." expression.

4. CONCLUSION

Specification techniques have received some attention in the recent years, although probably not enough. The term "structured programming" is too often associated with a method for writing good programs, or for writing them more easily. We think that software reliability involves not only the "structured writing" of programs, but more generally their "structured design". We have reasons to expect that the development of formal design methodologies is one of the best approaches for achieving this goal, and formal specification languages are a necessary element of such methodologies.

The design of specification languages presents some interesting questions, and the issues involved are somewhat different from those encountered in the design of programming languages. The language designer has to show a great concern for the potential users in order to make his language practical: there is a notion of cost involved in that specifications are only a part of the programming process: if the specification language is not easy to use (this implies both ease of writing and ease of reading), then the cost of using formal specifications will have a disproportionate part in the final cost of the software product. On the other hand, constraints have to be imposed for the sake of precision, so that the design of a specification language requires a subtle balance of conflicting elements. It is precisely the lack of an abundant literature describing past experience in this field that motivated this paper.

Although we have voluntarily limited ourselves to formal specifications of large software systems, it is our hope that the ideas presented above can still be of some interest in slightly different contexts.

ACKNOWLEDGMENTS

The author is indebted to several people who, by stimulating discussions or criticisms, have had a significant impact on this work: R. Boyer, R. Shostak and J Moore provided helpful comments on the problems created by the use of various mathematical formalisms in automatic verification. The experience of P. Griffiths in the design of a specification language was also of great interest, and the work of L. Robinson and several members of the Computer Science Laboratory at SRI in the use of a formal methodology and specifications was at the origin of this paper. In addition, the paper has greatly benefited from the comments of K. Levitt.

REFERENCES

[Boyer] : Boyer, R. S. and Moore, J S.: "Proving Theorems about LISP Functions," JACM, vol. 22, No. 1 (January 1975).

[Cook] : Cook, S. A. and Oppen, D. C.: "An Assertion Language for Data Structures," Proc. 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, California (January 20-22, 1975).

[Dahl] : Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R.: "Structured Programming," Academic Press (1972).

[Darlington] : Darlington, J. L.: "Automatic Program Synthesis in Second-Order Logic," Proc. Third International Joint conference on Artificial Intelligence, Stanford University, Stanford, California (August 20-23, 1973).

[Earley] : Earley, J.: "Toward an Understanding of Data Structures," Comm. ACM, Vol. 14, No. 10 (October 1971).

[Elspas] : Elspas, B., Levitt, K. N., Waldinger, R. J., and Waksman, A.: "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Vol. 4, No. 2 (June 1972).

[Ernst] : Ernst, G. W., and Hookway, R. J.: "The Use of Higher Order Logic in Program Verification," IEEE Transactions on Computers, Vol. C-25, No. 8 (August 1976).

[Flon] : Flon, L.: "Program Design with Abstract Data Types," Technical Report, Carnegie-Mellon University, Pittsburgh, Pennsylvania (June 1975).

[Floyd] : Floyd, R. W.: "Assigning Meaning to Programs," in Mathematical Aspects of Computer Science, Vol. 19, American Mathematics Society (1967).

[Griffiths] : Griffiths, P.: "SYNVER: A System for the Automatic Synthesis and Verification of Synchronization Processes," Proc. ACM 74 (November 1974).

[Guttag] : Guttag, J., Horowitz, E., and Musser, D.: "The Design of Data Structure Specifications," Proc. Second International Conference on Software Engineering, San Francisco, California (October 13-15, 1976).

[Hoare] : Hoare, C. A. R.: "An Axiomatic Basis for Computer Programming," Comm. ACM, Vol. 12, No. 10 (October 1969).

[Kowalski] : Kowalski, R.: "Predicate Logic as a Programming Language," Memo No. 70, Department of Computational Logic, University of Edimburgh, U.K. (November 1973).

[Liskov 1] : Liskov, B. H., and Zilles, S.: "Programming with Abstract Data Types:" Proc. ACM Conference on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4 (April 1974).

[Liskov 2] : Liskov, B. H., and Zilles, S.: "Specification Techniques for Data Abstractions," Proc. International Conference on Reliable Software, Los Angeles, California (April 1975).

[Manna 1] : Manna, Z., Ness, S., and Vuillemin, J.: "Inductive Methods for Proving Properties of Programs," Comm. ACM, Vol. 16, No. 8 (August 1973).

[Manna 2] : Manna, Z., and Waldinger, R. J.: "Toward Automatic Program Synthesis," Comm. ACM, Vol. 14, No. 3 (March 1971).

[Neumann] : Neumann, P. G., et al.: "On the Design of a Provably Secure Operating System," Proc. International Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France (August 1974).

[Parnas] : Parnas, D. L.: "A Technique for Software Module Specification with Examples," Comm. ACM, Vol. 15, No. 5 (May 1972).

[Robinson 1] : Robinson, L., and Levitt, K. N.: "Proof Techniques for Hierarchically Structured Programs," (to appear in Comm. ACM).

[Robinson 2] : Robinson, L., Levitt, K. N., Neumann, P. G., and Saxena, A. R.: "On Attaining Reliable Software for a Secure Operating System," Proc. International Conference on Reliable Software, Los Angeles, California (April 1975).

[Robinson 3] : Robinson, L., and Roubine, O. M.: "SPECIAL -- A SPECIfication and Assertion Language," (submitted to the Conference on Language Design for Reliable Software).

[Schwartz] : Schwartz, J. T.: "On Programming, an Interim Report on the SETL Project," Computer Science Department, Courant Institute of Mathematical Science, New York University (1973).

[Spitzen] : Spitzen, J. M., and Wegbreit, B.: "The Verification and Synthesis of Data Structures," _Acta Informatica_, Vol. 4 (1975).

[Tennent] : Tennent, R. D.: "The Denotational Semantics of Programming Languages," _Comm. ACM_, Vol. 19, No. 8 (August 1976).

[Waldinger] : Waldinger, R. J., and Levitt, K. N.: "Reasoning about Programs," _Artificial Intelligence Journal_, Vol. 5, No. 3 (Fall 1974).

[Warren] : Warren, H. S.: "Data Types and Structures for a Set Theoretic Programming Language," IBM Technical Report RC 5567 (23124) (August 1975).

[Wegner] : Wegner, P.: "The Vienna Definition Language," _Computing Surveys_, Vol. 4, No. 1 (March 1972).

[Wirth] : Wirth, N.: "Program Development by Stepwise Refinement," _Comm. ACM_, Vol. 14, No. 4 (April 1971).

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) The Design and Use of Specification Languages,		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report,
		6. PERFORMING ORG. REPORT NUMBER Technical Report CSL-47
7. AUTHOR(s) Olivier Roubine (10)		8. CONTRACT OR GRANT NUMBER(s) (15) N00123-76-C-0195
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford Research Institute, Computer Science Lab. 333 Ravenswood Avenue Menlo Park, CA 94025		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Part of deliverable A006
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 95152 (12) 20p.		12. REPORT DATE (11) 3 Oct. 1976 — 13. NO. OF PAGES 17
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited

(14) CSL-48

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Specification, Abstraction, Proof of Correctness, Formal Semantics, Design, Mathematical Logic, Verification, Software Reliability

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The goal of proving the correctness of computer programs has been hindered by the difficulty of stating formally what is to be proven. The tools developed to achieve this latter purpose often receive the name of "specification languages," but the formalization and implementation of such languages has seldom been carried out, and very few of these languages have been described in the literature.

We present some design issues concerning specification languages, e.g., desirable properties of such languages, and their limitations, as well as some ideas on

Unclassified 410 142
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)
20. ABSTRACT (Continued)

their use (and hence on the use of specifications in general) as an integral part of the design process leading to reliable software systems. The (possibly automated) verification of some properties of formal specifications is considered, and some conclusions are drawn concerning the help such an approach can bring in providing greater confidence in the final software product.

DD FORM 1 JAN 73 1473 (BACK)
EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DISTRIBUTION LIST

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Mr. Tony Allos Code 6201 Naval Ocean Systems Center 271 Catalina Boulevard San Diego, CA 92151	1 copy
Mr. L. Sutton Code 5200 Naval Ocean Systems Center 271 Catalina Boulevard San Diego, CA 92151	35 copies
Mr. William Carlson Advanced Research Projects Agency Office of Secretary of Defense 1400 Wilson Boulevard Arlington, VA 22209	15 copies
Mr. Neal Hampton Code 5200 Naval Ocean Systems Center 271 Catalina Boulevard San Diego, CA 92151	15 copies
Professor Stuart Madnick MIT - Sloan School E53-330 Cambridge, Mass 02139	1 copy
Mr. John Machado The Naval Electronic Systems Command National Center No. 1 Crystal City, Washington, D.C. 20360	5 copies