The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 251 July, 1983

FORMALIZING REUSABLE SOFTWARE COMPONENTS

Charles Rich & Richard C. Waters

ABSTRACT

There has been a long-standing desire in computer science for a way of collecting and using libraries of standard software components. Unfortunately, there has been only limited success in actually doing this. We believe that the lack of success stems not from any resistance to the idea, nor from any lack of trying, but rather from the difficulty of choosing an appropriate formalism for representing components. In this paper we define five desiderata for a good formalization of reusable software components and discuss many of the formalisms which have been used for representing components in light of these desiderata. We then briefly describe a formalism we are developing — the Plan Calculus — which seeks to satisfy these desiderata by combining together the best features of prior formalisms.

This paper has been accepted by the ITT Workshop on Reusability in Programming, Newport RI, September 7-9, 1983.

THE WIDE VARIETY OF COMPONENTS

The biggest problem with developing an appropriate formalization for components is that there are many different kinds of commonalities between programs which it would be beneficial to express as reusable components. To illustrate the diversity of components we will focus on the following six examples:

Matrix add - The algorithm for adding together two matrices. It should be specified independent of the data representation which is used for the matrices and the type of quantities which are stored in the matrices.

Stack - The stack data structure and its associated operations PUSH and POP. The representation and operations should be specified independent of the type of quantities to be stored in the stack.

Filter positive - The idea of selecting the positive elements of a temporal sequence of quantities available in a loop. For example, in the following fragmentary loop, the IF implements a *filter positive.*

```
DO ...
  X = ...
  IF X>0 THEN ... X ...
END
```

This component should be specified independent of the way the sequence of quantities is created and the way the selected quantities are to be used, as well as independent of the type of the quantities in the series. In addition, the specification should be independent of whether the loop will be implemented iteratively, as above, or recursively.

Master file system - The idea of a cluster of programs (report programs, update programs, audit programs, etc.) which operate on a single *master* file which is the sole repository for information about some topic. This component is essentially just a set of constraints on the programs and how they interact with the file. It should be specified independent of the data to be stored in the file and the actual computation to be performed by the programs.

Deadlock free - The idea that a set of asynchronously interacting programs are designed so that they cannot reach a state where each program is blocked waiting for other programs to act. This component places restrictions on the ways in which the programs can interact. It is independent of the computations to be performed by the programs.

Move invariant - The idea that the computation of an expression can be moved from inside of a loop to outside of the loop as long as it has no side-effects and all of the values it references are constants within the scope of the loop. It should be specified independent of the computation being performed by the expression, and of other computation in the loop.

The example components above differ from each other along many dimensions. *Matrix add* is primarily a computational component which specifies a particular combination of operations, while *stack* is a data component which primarily specifies a particular combination of data objects. *Matrix add* and *stack* also differ in that *matrix add* is a concrete algorithm while *stack* is much more of an abstract concept. Another dimension of difference between components is that while *matrix add* can be used in a program as a simple subunit, *filter positive* is much more fragmentary and must be combined together with other loop fragments in order to perform a useful computation.

In contrast to the first three components which are low-level, localized units, both *master file system* and *deadlock free* are high-level, diffuse concepts which correspond more closely to sets of constraints than to computational units. These two components in turn differ in that *master file system* is a relatively straightforward set of constraints which can be satisfied individually, while *deadlock free* is a property of an entire system of programs which critically depends on each detail of the interaction between the programs. *Move invariant* differs from all of the other components in that it is an optimization which is achieved by a standardized transformation *on* programs rather than a standardized computation performed *by* programs.

DESIDERATA FOR A FORMALIZATION

Many properties are required of a formalization in order for it to be an effective representation for reusable components. The following five desiderata stand out as being of particular importance.

Expressiveness - The formalism must be capable of expressing as many different kinds of components as possible.

Convenient combination - The methods of combining components must be easy to implement and the properties of combinations should be evident from the properties of the parts.

Semantic soundness - The formalism must be based on a mathematical foundation which allows correctness conditions to be stated for the library of components.

Machine manipulability - It must be possible to manipulate the formalism effectively using computer tools.

Programming language independence - The formalism should not be dependent on the syntax of any particular programming language.

Given the wide range of components which it would be useful to represent, the expressiveness of a formalization is paramount. An important, though hard to assess, aspect of this is *convenience.* It is not sufficient that a formalism merely be *capable* of representing a given component. To be truly useful, the formalization must be able to conveniently represent the component in a straightforward way which supports the other desiderata rather than representing it via a circumlocution which impedes the other desiderata.

Convenient combination properties are also essential, since they are the way in which components are in fact reused. An important part of this is the desire for *fine granularity* in the representation. The goal is to have each component embody only a single idea or design decision so that the user has the maximum possible freedom to combine them as he chooses.

A firm semantic basis is needed for a good formalization so that it is possible to be certain of what is being represented by a given component, and so that the combination process preserves the key properties of components. Note that the semantic basis does not necessarily need to make totally automatic verification possible. Though less convenient, manual or machine-aided verification of library components is sufficient in many situations.

Machine manipulability of a formalization is a key issue. Tools need to be developed to support the automatic conversion of a combination of components into a program. In addition, there are thousands of programming ideas which it would be useful to express as components. In order to be able to effectively deal with such a large library, automatic aids need to be developed to assist the user in creating, modifying, selecting, and combining components.

A major problem with previous formalisms has been the focus on existing programming languages as the basis for defining reusable components, when there are in fact important differences between the original goals of these programming languages and the goals of the reusable components work. Existing programming languages were designed primarily to express complete programs in a form which is easily readable by the programmer and which can be effectively executed by a machine. In contrast, the challenge in reusability is to express the fragmentary and abstract components out of which complete programs are built.

There are two additional reasons behind the desire for language independence. First, when expressing a component one does not want to be unnecessarily specific about superficial language details. For example, when specifying the PUSH operation for a stack, one does not want to have to specify particular variable names, or whether the operation is to be coded in-line or out-of-line when it is used. Second, the vast majority of components have nothing to do with any particular programming language. Expressing them in a particular programming language only limits their applicability.

APPROACHES TO FORMALIZATION

The following sections discuss a number of approaches to the formalization of components. The relative strengths and weaknesses of the approaches are evaluated in the light of the five desiderata presented above. The central theme which ties the sections together is the search for formalisms that are capable of expressing the wide range of components desired without sacrificing the other desiderata.

As a point of comparison for other formalisms, one must consider free-form English text. Much of the knowledge we seek to formalize is already captured informally in the vocabulary of programmers and in text books on programming [1,16]. The great strength of English text is expressiveness. It is capable of representing any kind of component. Moreover, it is programming language independent. Unfortunately, English text does not satisfy any of the other desiderata presented above. There is no theory of how to combine textual fragments together; there is no semantic basis that makes it possible to determine whether or not a piece of English text means what you think it means; and free-form English text is not machine manipulable in any significant way.

Subroutines

Subroutines have many advantages as a representation for components. They can be easily combined by writing programs which call them. They are machine manipulable in that high level language compilers and linkage editors directly support their combination. Subroutines are very much programming language dependent. However, this gives them a firm semantic basis via the semantics of the programming language they are written in.

Unfortunately, subroutines are limited in their expressiveness. They are really only convenient for expressing localized computational algorithms such as *matrix add*. They cannot represent data components such as *stack*, fragmentary components such as *filter positive*, diffuse high level components such as *master file system*, or transformational components such as *move invariant*. In addition they lack fineness of granularity. It is difficult to write a subroutine without gratuitously specifying numerous details that are not properly part of the component. For example, in most languages there is no convenient way to write a subroutine representing *matrix add* without specifying the data representation for the matrices and the numbers in them.

Macros

A subroutine specifies a fixed piece of program text corresponding to a component. The only variability allowed is in the arguments which are passed to the subroutine in a given call on the subroutine. In contrast, a macro specifies an arbitrary computation which is used to create a piece of program text corresponding to a use of a component. Due to the provision for arbitrary computation, macros are a considerable improvement over subroutines in expressiveness. They can be used to represent data components and fragmentary components. In addition, they can represent components at a much finer granularity. For example, it is straightforward to write a macro which represents *matrix add* independent of the data structures it operates on. Note however, that macros are still not suited to representing diffuse components or transformational ones.

Like subroutines, macros are machine manipulable in that macro processors directly support the evaluation of macro calls, and the integration of the resulting program text into the program as a whole. Unfortunately, macros are less satisfactory than subroutines in other respects. Though macro calls are combined syntactically in essentially the same way as subroutine calls, their combination properties are not as simple. For example, since a macro can perform arbitrary computation utilizing its calling form in order to create the resulting program text, there is no guarantee that nested macro calls will be allowed to operate as they were intended. The macro writer must take extreme care in order to insure that flexible combination is possible. This unfortunately militates against the increased expressiveness which is the primary advantage of macros.

The paramount problem with macros is that they lack a firm semantic basis. Because they allow arbitrary computation, it is very difficult to verify that a macro accurately represents a given component. It is even more difficult to show that a pair of macros can be combined without destructive interaction.

Program Schemas

There has been a considerable amount of theoretical investigation of program schemas as a vehicle for representing components [3,9,21,22,23,29]. Program schemas are essentially templates with holes in them which can be filled in with user supplied program text. As such, they can be viewed as a compromise between subroutines and macros. The main improvement of program schemas over macros is that, like subroutines, they have a firm semantic foundation in the semantics of the programming language they are written in, and their combination properties are relatively straightforward.

There has not been very much activity directed towards creating an actual programming environment incorporating a library of program schemas. However, there is no reason to believe that program schemas are not at least as machine manipulable as macros. For example, one could create a programming environment supporting program schemas by taking a standard macro processor and limiting the macros that could be written to ones which were essentially program schemas.

Unfortunately, though program schemas are an improvement in expressiveness over subroutines, they are significantly less expressive than macros. Program schemas are of some use in representing data components such as *stack*, and can represent components at a finer granularity than subroutines. Like macros they could be used to conveniently represent *matrix add* independent of the data structures it operates on. However, unlike macros, program schemas cannot in general be used to represent fragmentary components such as *filter positive*. Going beyond this, they are no more useful than macros at representing diffuse or transformational components.

Flowcharts and Flowchart Schemas

A limitation of subroutines, macros, and program schemas is that they are fundamentally programming language dependent. This blocks the transfer of components between languages. More importantly, it forces components to be represented in terms of specific control flow and data flow constructs which may not be an essential part of the component and which may limit the way in which they can be combined.

One way to alleviate this problem would be to write components in a programming language independent representation such as a flowchart. Flowcharts use boxes and control flow arrows in order to specify control flow independent of any particular control flow construct. Similarly, data flow arrows can be used to represent data flow independent of any particular data flow construct [7].

A flowchart is basically equivalent to a subroutine, and has the same level of expressiveness. In analogy to program schemas one can gain additional expressiveness by using flowchart schemas [15,19] which are flowchart templates with holes in them where other flowcharts can be inserted. Just as a programming language can be given a rigorous semantic foundation, a flowchart language can be given a semantic foundation which will serve as a semantic basis for components. In addition, flowcharts and flowchart schemas can be combined together in basically the same semantically clean ways that subroutines and program schemas can be.

To date, flowcharts have primarily been used as a documentation and design aid and have not been given much machine support. However, there is no reason why they cannot be represented in a machine manipulable form and used as part of a programming environment. The Knowledge-Based Editor system [28] demonstrates the feasibility of this concept. The only significant difficulty stems from the need for a translation module which can convert flowcharts into program text in order to interface with the rest of the programming environment.

Flowcharts and flowchart schemas are a significant improvement over subroutines and program schemas in that they are programming language independent. However, with regard to the other desiderata, there are basically identical. In particular, they are no more expressive. As a result, they are still not really satisfactory as a representation for reusable components.

Logical formalisms

All of the formalisms above can be thought of as *algorithmic* in that they share the following approach to the problem of representing components: They represent a component by giving an example (or template) of it in a programming (or flowchart) language. In addition, the only way to use a component is to place it somewhere in a program. This fundamentally limits the expressiveness of these formalisms. They can represent only localized algorithmic components because the languages being used are only capable of representing algorithms, and the way the components are used requires them to be localized.

The extensive work on specifying the semantics of programming languages suggests a completely different approach to the problem of specifying components: using logical formalisms (e.g., the predicate calculus) to represent components. A key advantage of logical formalisms is semantic soundness. In fact, logical formalisms provide the semantic basis for all of the formalisms presented in this paper. We have already seen that, in the role of providing a semantic basis for programming languages, logical formalisms provide the semantic basis for the algorithmic formalisms presented above. An implicit part of this is that logical specifications must be provided for components so that they can be verified (by hand if necessary).

Another important advantage of logical formalisms is in the area of expressiveness. In contrast to the algorithmic formalisms, logical formalisms have no trouble representing diffuse high level components such as *master file system* and *deadlock free.* The usefulness of such components is enhanced by the fact that logical formalisms also have very convenient combination properties. Specifically, the theory generated by the union of two axiom systems is always either the union of the theories of the two component systems or a contradiction, but never some third, unanticipated theory. An additional advantage of logical formalisms is that they are inherently programming language independent.

However, logical formalisms are quite cumbersome when it comes to specifying algorithmic components such as *matrix add* (as opposed to the specifications for algorithmic components). Given a component such as *stack,* which combines some non-algorithmic aspects with some algorithmic aspects, logical formalisms are convenient for the former, but not the latter. Both of the above suggest that logical formalisms are best used as an adjunct to, rather than a replacement for, algorithmic formalisms. It should be noted that neither logical nor algorithmic formalisms are particularly well suited to representing transformational components such as *move invariant.*

The great weakness of logical formalisms is in the area of machine manipulability. It is not hard to represent logical formulas in a machine manipulable way. However, it is hard to do very many useful things with them. The key difficulty is that, at the current state of the art, only simple logical deductions are possible in practical applications. For example, if a programming system were to be based on the combination of components represented by logical formulas, a component would be required which could create program text corresponding to sets of logical formulas. Unfortunately, attempts to produce program text from logical descriptions have not been practical to date.

This problem again suggests that it might be fruitful to combine logical and algorithmic formalisms in order to reduce the amount of deduction which must be performed. Unfortunately, it is not clear how this can be helpful with regard to components such as *master file system* and *deadlock free* which have no algorithmic aspects. Assumedly, if one includes *deadlock free* as one of the components describing a set of programs one would like the programming system to be of some assistance in producing programs which are safe from deadlock, or at the very least, be able to detect when deadlock is possible. However, it is not clear that even the latter goal is achievable given the current state of the art of automatic theorem proving.

Data Abstraction

An interesting area of inquiry which has combined logical and algorithmic formalisms is data abstraction. A considerable amount of research has been done on how to state the specifications for a data structure and its associated access functions [10,13,14,17]. This provides a semantic basis for data abstractions and for methods of combining them. In addition, languages such as Alphard [30], CLU [18], and ADA have been developed which have constructs which directly support the specification of data components such as *stack.* This demonstrates the ease with which data abstractions can be represented in a machine manipulable (though language dependent) form.

The contribution of data abstractions is that they extend the expressiveness of algorithmic formalizations into the realm of components with data structure aspects. For example, generic ADA packages make it possible to represent *stack* in full generality. They also make it possible to represent *matrix add* independent of the data representation which is used for the matrices.

Program Transformations

Another way of representing components is as program transformations [2,4,5,6,8,12,27]. A transformation matches against some section of program text (or more usually its parse tree) and replaces it by a new section of program text (or parse tree). A typical transformation has three parts. It has a pattern which matches against the program in order to determine where to apply the transformation. It has a set of logical applicability conditions which further restrict the places where the transformation can be applied. Finally, it has a (usually procedural) action which creates the new program section based on the old section. Note that when applied to small localized sections of a program, program transformations are very much the same as macros.

An important aspect of program transformations is the idea of a *wide spectrum language.* In contrast to ordinary high level languages, wide spectrum languages contain syntactic and semantic extensions which are not directly executable. In some cases these higher level constructs have a semantics independent of the transformation system, but often they are defined only in terms of the transformations which convert them into executable constructs.

The most interesting contribution of transformations is that they view program construction as a *process.* Rather than viewing a program solely as a static artifact which may be decomposed into components the way a house is made up of a floor, roof and walls, transformations view a program as evolving through a series of construction steps which utilize components which may not be visible in the final program, just as the construction of a house actually requires the use of scaffolding and other temporary structures. This point of view enables transformations to express components such as *move invariant* which are common steps in the construction of a program rather than common steps in the execution of a program.

Another important aspect of transformations is that they can be combined in a way which is quite different from the other formalisms. As mentioned above, many simple transformations are basically just macros which specify how to implement particular high level constructs in a wide spectrum language. These transformations are only triggered when instances of their associated high level constructs appear; thus they only operate where they are explicitly requested and combine in essentially exactly the same way as macros.

However, other transformations are much less localized in the way the operate. For example, a transformation representing *move invariant* would have applicability conditions (e.g., that the expression is invariant) which must look at large parts of the program. In addition, such transformations are not intended to be applied only when explicitly requested by the user. Rather, they are intended to be used whenever they become applicable for any reason. This makes powerful synergistic interaction between transformations possible.

If transformations are allowed to contain arbitrary computation in their actions, they have the same difficulty with regard to semantic soundness and convenient combination that macros have. The transformation writer has to take great care in order to insure that the interaction between transformations will in fact be synergistic rather than antagonistic. In order to have a semantic basis, transformations must include a logical description of what the transformation is doing. One important way that this has been done is to focus on transformations which are correctness preserving — ones which from a logical perspective do nothing.

A number of experimental systems have been developed which demonstrate that transformations are machine manipulable. These systems support the automatic selection of which transformations to apply as well as their actual application.

A problem with transformations is that, as generally supported, they are very much programming language dependent. This not only limits the portability of components represented as transformations, it also limits the way transformations can be stated by requiring that every intermediate state of a program being transformed has to fit into the syntax of the programming language. One way to alleviate these problems would be to apply transformations to a programming language independent representation such as flowcharts.

THE PLAN CALCULUS

As part of the Programmer's Apprentice project [24,28], we have developed a formalism, called the Plan Calculus [25], which seeks to satisfy the five desiderata defined above by combining ideas from flowchart schemas, data abstraction, logical formalisms, and program transformations.

In the Plan Calculus, components are represented as *plans.* In order to achieve language independence, the algorithmic aspects of plans are represented as flowchart schemas. In these flowcharts, computations are represented as boxes with input and output ports. Both the control flow and data flow between the boxes is represented using explicit arcs. As a result, the flowcharts do not depend on any particular control flow or data flow constructs. The flowcharts are hierarchical — a box in a flowchart can contain an entire sub-flowchart. The flowcharts are schematic — they can have empty boxes (called *roles*) which will be filled in later.

The flowchart schema part of a plan is annotated with several kinds of logical assertions. Each box is specified by a set of preconditions and postconditions. Logical constraints between roles can also be specified in order to limit the way in which the roles are to be filled in. Finally, for hierarchically nested flowcharts, a network of dependency links records a summary of the proof that the specifications of the outer box follow from the combination of the specifications of the inner boxes. Components such as *master file system* and *deadlock free* which have little or no algorithmic aspect are represented by plans which consist almost entirely of assertions with little or no flowchart information.

In order to unify the concept of a plan for an algorithm with the concept of a plan for a data structure, the basic flowchart representation is extended so that it can contain components which correspond to data as well as sub-computations. Data parts can be left unspecified as data roles. The same kinds of logical annotations are applied to the data parts of a plan as to the computational parts of plans. Given these extensions, plans are capable of representing the same kind of information as data abstraction mechanisms. As an example, the plan for *stack* consists of a number of logically interrelated flowchart schemas, one of which represents the stack data object and the rest of which represent the operations on the stack.

Transformational components such as *move invariant* are represented by *overlays*. An overlay is a mapping between a two plans. It specifies a set of correspondences between the roles of the plans. Overlays can be thought of as transformations in which both the left and right hand side are plans. They differ from program transformations, however, in that they can be used like grammar rules for both analysis and synthesis.

The logical assertions in a plan are stated directly as predicate calculus formulas. Overlays are formally functions on plans. The flowchart schemas in plans are given a semantic foundation by defining them in terms of a version of the situational calculus [11]. All of the features of the flowcharts — the boxes, ports, control flow arrows and data flow arrows — are directly translated into situational calculus assertions. As a result, any given flowchart can be viewed as the abbreviation for a set of situational calculus assertions. Manna and Waldinger have recently used the situational calculus in a similar way in order to specify certain problematic features of programming languages [20].

Since the situational calculus is essentially just predicate calculus with some conventions applied, everything in a component (flowchart information, assertions, and overlays) can be reduced to a set of logical axioms. This reduction has several important consequences from the point of view of the desiderata stated in this paper. First, since essentially any component can be expressed by some set of axioms, essentially any component can be expressed by a plan. Note however that from the standpoint of practicality, it is very important that most components can in fact be represented primarily in terms of flowchart schemas and overlays rather than merely as a set of axioms. Second, the combination of two plans amounts semantically to the union of axioms, which has the desirable properties described earlier. Third, everything in a plan is machine manipulable given a reasoning system which is capable of performing logical deductions.

Unfortunately, as mentioned in the section on logical formalisms, current general purpose reasoning systems are not powerful enough to be of practical use. As a result, in order to make plans machine manipulable in a practical sense, we have developed a number of special purpose modules for operating on plans. First, we have developed a coder module which converts a plan into program text, and an analysis module which converts program text into a plan [28].

More fundamentally, the Plan Calculus has been designed to combine algorithmic and logical formalisms in order to support a *layered approach* to reasoning. The goal is to replace complex logical reasoning with graph-theoretic operations on algorithmic representations leaving only simple deductions to be performed in the logical domain. We have designed a reasoning module called CAKE [26] which supports this kind of reasoning. It does as much deduction as possible by operating directly in terms of flowcharts schemas and overlays, falling back on weak general methods for any additional deduction. For example, combination of flowchart schemas is actually performed by substituting one graph into the other. Simple general reasoning is used in order to determine whether or not the relevant preconditions and constraints permit the substitution to take place.

One consequence of this layered approach to reasoning is that it is very important to have as much information as possible represented in terms of flowchart schemas and overlays so that it can be efficiently processed. It should be noted that components such as *deadlock free* which have little or no algorithmic aspect and which require complex reasoning cannot be effectively handled by the Plan Calculus representation at the current time. They can be represented, but the reasoning system is not powerful enough to make practical use of them.

In closing, we summarize how the Plan Calculus seeks to meet the five desiderata for a formalism for representing reusable components.

Expressiveness - Plans combine a number of different representations in order to be able to express a wide range of components. The algorithmic aspects of components are represented in terms of flowchart schemas. The assertional aspects of components are directly represented in terms of predicate calculus assertions. The transformational aspects of components are represented in terms of overlays.

Convenient combination - Sets of predicate calculus assertions can be combined by simply asserting both thereby obtaining their union. Flowchart schemas can be combined by graph-theoretic operations. This is facilitated by the fact that everything is represented locally in the flowchart schemas so that combining two schemas cannot cause them to interfere with each other. Overlays, like transformations, can be combined by applying them one after another.

Semantic soundness - The logical assertions in a plan are expressed directly in the predicate calculus. The flowchart schemas are given a semantic basis by formally defining them as abbreviations for sets of situational calculus assertions. Overlays are defined as functions on plans.

Machine manipulability - All of the plan constituents (predicate calculus assertions, flowchart schemas, and overlays) are represented in machine manipulable forms. Plans for components are stored in a plan library indexed according to their specifications, and according to their relationships with each other. Modules have been implemented which construct program text corresponding to a plan and construct a plan corresponding to a section of program text. Reasoning about plans is efficiently supported by a layered reasoning system which can perform simple deductions by reasoning directly in terms of predicate calculus assertions and can perform much more complicated deductions by reasoning in terms of flowchart schemas and overlays.

Programming language independence - Plans and overlays are composed solely of flowchart schemas and logical formulas, both of which are inherently programming language independent.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] R. Balzer, "Transformational Implementation: An Example", *IEEE Trans. on Software Eng.*, Vol. 7, No. 1, January, 1981.

[3] S. Basu and J. Misra, "Some Classes of Naturally Provable Programs", *2nd Int. Conf. on Software Eng.*, San Francisco, Cal., Oct., 1976.

[4] M. Broy and P. Pepper, "Program Development as a Formal Activity", *IEEE Trans. on Software Eng.*, Vol. 7, No. 1, January, 1980.

[5] R.M. Burstall and J.L. Darlington, "A Transformation System for Developing Recursive Programs", *J. of the ACM*, Vol. 24, No. 1, January, 1977.

[6] T.E. Cheatham, "Program Refinement by Transformation", *5th Int. Conf. on Software Eng.*, San Diego, Cal., March, 1981.

[7] J.B. Dennis, "First Version of a Data Flow Procedure Language", *Proc. of Symposium on Programming*, Institut de Programmation, U. of Paris, April 1974, pp. 241-271.

[8] R.B. Dewar, M. Sharir, E. Weixelbaum, "Transformational Derivation of a Garbage Collection Algorithm", *ACM Trans. on Programming Languages and Systems*, Vol.4, No.1, pp.650-667, October, 1982.

[9] S.L. Gerhart, "Knowledge About Programs: A Model and Case Study", in *Proc. of Int. Conf. on Reliable Software*, June 1975, pp. 88-95.

[10] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology, Vol. IV*, (ed. Raymond Yeh), Prentice-Hall, 1978.

[11] C. Green, "Theorem Proving by Resolution as a Basis for Question-Answering Systems, *Machine Intelligence 4*, D. Michie and B. Meltzer, Eds., Edinburgh University Press, Edinburgh, Scotland, 1969.

[12] C. Green and D. Barstow, "On Program Synthesis Knowledge", *Artificial Intelligence*, Vol. 10, No. 3, November 1978, pp. 241-281.

[13] J. Guttag, "Abstract Data Types and the Development of Data Structures", *Comm. of the ACM*, Vol. 20, No. 6, June 1977, pp. 396-404.

[14] J.V. Guttag, E. Horowitz and R.D. Musser, "Abstract Data Types and Software Validation", *Comm. of the ACM*, Vol. 21, No. 12, pp. 1048-1064, December, 1978.

[15] Y.I. Ianov, "The Logical Schemes of Algorithms," in *Problems of Cybernetics*, Vol. 1, Pergamon Press, New York (English Translation), pp. 82-140.

[16] D.E. Knuth, *The Art of Computer Programming*, Vol. 1,2,3, Addison-Wesley, 1968,1969,1973.

[17] B.H. Liskov and S.N. Zilles, "An Introduction to Formal Specifications of Data Abstractions," *Current Trends in Programming Methodology, Vol. I*, (ed. Raymond Yeh), Prentice-Hall, 1977.

[18] B. Liskov *et. al.*, "Abstraction Mechanisms in CLU", *Comm. of the ACM*, Vol. 20, No. 8, August 1977, pp. 564-576.

[19] Z. Manna, *Mathematic Theory of Computation*, McGraw-Hill, 1974.

[20] Z. Manna and R. Waldinger, "Problematic Features of Programming Languages: A Situational-Calculus Approach; Part I: Assignment Statements", Stanford Univ., Weizmann Institute and the Artificial Intelligence Center, August, 1980.

[21] J. Misra, "A Technique of Algorithm Construction on Sequences", *IEEE Trans. on Software Eng.*, Vol. 4, No. 1, pp. 65-69, January, 1978.

[22] J. Misra, "An Approach to Formal Definitions and Proofs of Programming Principles", *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 5, September 1978, pp. 410-413.

[23] J. Misra, "Some Aspects of the Verification of Loop Computations", *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 6, November 1978, pp. 478-485.

[24] C. Rich, H.E. Shrobe, and R.C. Waters, "An Overview of the Programmer's Apprentice", *Proc. of 6th Int. Joint Conf. on Artificial Intelligence*, Tokyo, Japan, August, 1979.

[25] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981.

[26] C. Rich, "Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too", *Proc. of Second National Conf. on Artificial Intelligence*, Pittsburgh, PA, August, 1982.

[27] T.A. Standish, D.C. Harriman, D.F. Kibler, and J.M. Neighbors, *The Irvine Program Transformation Catalogue*, U. of Cal. at Irvine, 1976.

[28] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Trans. on Software Eng.*, Vol. SE-8, No. 1, January 1982.

[29] N. Wirth, *Systematic Programming, An Introduction*, Prentice-Hall, 1973.

[30] W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs", *IEEE Trans. on Software Eng.*, SE-2, No. 4, December 1976, pp. 253-265.