A FORMAL REPRESENTATION FOR PLANS IN THE PROGRAMMER'S APPRENTICE

Charltt Rich

The Artificial Intelligence Laboratory[1]
Mattechutettt Institute of Technology
Cambridge, Massachusetts 02139

Abstract

A plan calculus is presented which is being used to represent programs and a library of standard data and control abstractions in the programmer's apprentice. Important features of this formalism include: programming language independence, additivity, verifiability and multiple points of view. The logical foundations of the representation are specified formally using a situational calculus in which side effects and overlapping mutable data structures *are* accounted tor. The plan calculus is comparedwith other formalisms, such as program schemas, and its advantages pointed out.

1. Introduction

this paper reports on recent developments in a formalism celled the *plan calculus* which is being used in the programmer's apprentice [27] to represent and reason about programs The plan calculus was originally developed by Rich and Shrobe [25] and was subsequently elaborated by Shrobe [34] and Waters [38]. Most recently the author has extended the plan calculus by adding data Abstraction and multiple points of view, as well as by providing a formal semantics.

The goal of the programmer's apprentice project is to develop a knowledge-based tool for program development The programmer's apprentice attacks the complexity barrier [41] in programming in several ways.

(i) by providing an on-line database of design decisions, documentation and descriptions of a program's design at various levels of detail;

(ii) by reasoning about these descriptions to detect inconsistencies and predict the influence of incremental changes;

(iii) and by providing a library of standard forms for use in program construction and analysis.

This paper focusses on the representation used in building the library of standard forms. Reasoning about plans will not be discussed in detail (see [34]), other than to describe the formal system in which such reasoning takes place; nor will the detailed contents of the library be discussed, other than to show some examples.

The utility of a library of standard forms in program development is motivated by the observation that expert programmers .(like experts in many other fields) *are* distinguished from novices by their use of a much richer vocabulary of intermediate level programming concepts The novice programmer thinks of constructing a program out of assignments, tests, arrays, DO loops, etc., whereas the expert programmer works with larger, less language-specific concepts, such as searching, accumulation, hashing, etc. There are deep reasons for this having to do with managing the complexity of the design process [26). To the extent that the programmer's apprentice can provide a library of such standard building blocks, the programmer benefits by a reduction in the amount he needs to say to construct a program

The idea of developing libraries of standard software components to gain the same benefits that the use of standard components has provided in other areas of engineering is not new [32]. I believe that part of the reason efforts in this direction have not been more successful is because of deficiencies in the formalisms used. The plan calculus is an attempt to remedy these deficiencies.

1. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-75-C-0643 and N00014-80-C-0505, and in part by National Science Foundation grant MCS-7912179 The views and conclusions contained in this paper are those of the author, and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Department of Defense, the National Science Foundation, or the United States Government.

2. The Plan Calculus

To a first approximation, the plan calculus may be thought of as unifying in one formalism ideas from flowchart schemas [18], data flow schemas [8], program transformations [6,1,5] and abstract data types [17] An example of a *plan* is shown in Fig. 1. The details of the notation used in this figure can wait until later; for now the reader may note that a plan is basically a hierarchical structure made up of different kinds of boxes and arrows. The inner rectangular boxes denote operations and tests, while the arrows between boxes denote data flow (solid arrows) and control flow (hatched arrows) The chief features of this representation, as compared with previous formalisms are as follow,

(i) *Language independence* To achieve some measure of canonical form, the plan calculus suppresses features of a program which stem soley from the way an algorithm must be expressed in a particular programming language.

(ii) *Additivity.* For a library to be easy to use, the rules for combining forms must be straightforward and explicit In the plan calculus, the result of combining two non-contradictory forms is always a form which satisfies the constraints of both of the original forms.

(iii) *Verifiability.* Simply having a library of standard forms addresses only the low productivity part of the software crisis. The plan calculus also provides leverage on the software reliability problem by providing a methodology (in this case, an axiomatic semantics) for verification of library components. Furthermore, this methodology provides a framework in which inconsistencies between uses of standard forms can be detected.

(iv) *Multiple Points Of View.* Multiple and overlapping points of view are represented in the plan calculus by *overlays* between plans. An example of an overlay is shown in Fig. 2 (to be discussed further in the next section) Overlays are used to express the relationship between levels of implementation, to describe overlapping module hierarchies, and to decompose data and control structures in ways which make their relationship to the

library more explicit.

An additional desideratum we set for representing the library of standard forms in the programmer's apprentice is that the formalism must be neutral between analysis, synthesis and verification of programs. This turns out to be of great practical importance for building an interactive programming aid, since in real program design situations all three of these activities *are* Intermingled. A neutral representation of standard forms is also theoretically more interesting since it is more likely a *priori* to capture significant features of human understanding of these forms than a representation tailored specifically for analysis, synthesis or verification only.

Comparison with Other Formalisms

Past efforts to construct knowledge bases for automatic or partially automated programming have used one of the following formalisms: program schemas [11], program transformations (6,1,5], program refinement rules [2], or formal grammars [30]. Although each of these representations has been found useful in certain applications, none combines all of the important features of the plan calculus listed above. This section will serve both to contrast the plan calculus with these other formalisms, and as the preliminary to a more careful definition of the plan calculus to follow.

For example, program schemas (incomplete program texts with constraints on the unfilled parts) have been used by Wirth [42] to catalog programs based on recurrence relations, by Basu and Misra [3] to represent typical loops for which the loop invariant is already known, and by Gerhart [11] and Misra [23] to represent and prove the properties of various other common forms. Unfortunately, the syntax of conventional programming languages is not well suited for the kind of generalization needed in this endeavor. For example, the idea of a search loop (a standard programming form) expressed informally in English should be something like the following.

A *search loop* is a loop with two exits in which a given predicate (the same one each time) is applied to a succession of objects until either the predicate is satisfied, in which case that object is made available for use outside the loop, or the objects to be searched are exhausted.

In Lisp, as in other languages, this kind of loop can be written in innumerable forms, many of which are syntactically (and structurally) very different, such as:

```
(prog ()
   LP (COND (exhausted (RETURN NIL)))
      (COND {{predicate current}(RETURN current)))
      (GO LP))
```

or with only one RETURN instead of two,

```
(PROG ()
   LP (COND (exhausted NIL)
            (T ...
               (COND ((predicate current)
                      (RETURN current)))
               (GO LP))))
```

or even recursively, e.g.

```
(DEFINE SEARCH ()
   (COND (exhausted NIL)
         (T ...
            (COND ((predicate current) current)
                  (T ...
                     (SEARCH))))))
```

The problem here is that conventional programming languages *ere* oriented towards specifying computations in enough detail so that a simple local interpreter can carry them out. Unfortunately a lot of this detail is often arbitrary and conceptually unimportant. In the plan calculus, all three of the schemas above (and many other such variations) are expressed by the single plan shown in Fig. 1.

A new generation of programming languages descended from Simula (7), such as CLU[16] and Alphard [33), provide a syntax for specifying standard forms such as the search loop in a more canonical way. However, there are two more fundamental difficulties with using program schemas to represent standard program forms, which Simula and its descendants do not solve. First, programs (and therefore program schemas) *are* not in general easy to combine, nor are they additive. This means that when you combine two program schemas, the resulting schema is not guaranteed to satisfy the constraints of both of the original schemas, due to such factors as destructive interactions between variable assignments. Second, existing programming languages do not allow multiple views of the same program or overlapping module hierarchies. I believe the reason for this is that a program

Figure |. Plan for a Search Loop

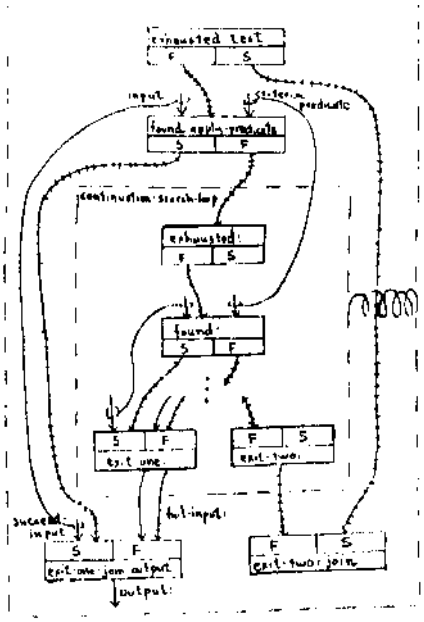
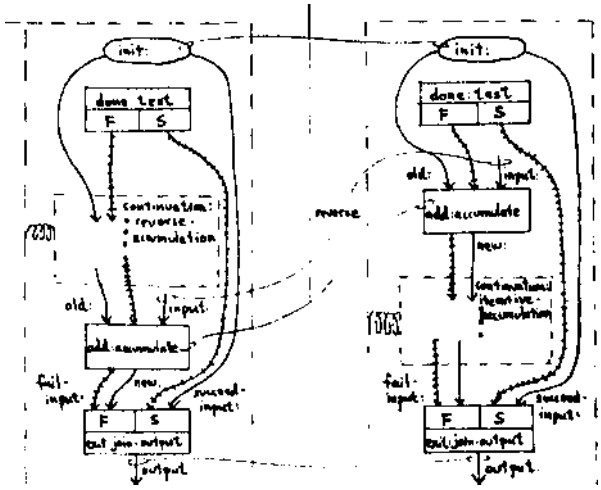


Figure 2. Overlay Between Accumulation on the Way Down vs. Up.



Rverse-accumulation *Iterative* - accumulation.

reverse> Iterative-accumulatio

is still basically thought of, from the standpoint of these languages, as a set of instructions to be executed, rather than as a set of descriptions (e.g. blueprints) which together specify a computation.

Currently the most common way to represent relationships between standard forms (typically implementation/abstraction relationships) is via program transformations or program refinement rules [2]. As compared to overlays, these formalisms have two serious problems which stem from their lack of neutralness between analysis and synthesis, An overlay in the plan calculus, as in Fig. 2, is made up of two plans and a set of *correspondences* between the parts of the two plans. Each plan represents a point of view; the correspondences express the relationship between the points of view. For example, in an Implementation overlay the plan on the right hand side is the abstract description and the plan on the left hand side is an implementation. It is important, however, that either plan can be used as the "pattern". In a typical program synthesis step using overlays the right hand plan is used as the pattern and the left hand plan is instantiated as a further implementation Conversely, in a typical analysis step, the left hand plan serves as the pattern and the right hand plan is instantiated as a more abstract description. With both program refinement rules and Knowledge-based' program transformations this sort of symmetric use is not possible since the right hand side of a transformation or refinement rule is typically a sequence of substitutions or modifications to be performed, rather than a pattern.

A second problem stemming from the asymmetry of program transformations and refinement rules is their lack of verifiability. The correctness of an overlay in the plan calculus is verified by proving essentially that the the constraints of the plan on the left hand side, together with the correspondences (which are formally a set of equalities between terms on the left and terms on the right) imply the constraints of the plan on the right hand side. Neither Balzer's transformation language nor Green and Barstow's refinement tree notation has been adequately formalized to permit the question of correctness to be addressed. The recent work of Broy and Pepper *[A]* is an improvement in this direction, since their transformations have program forms on both the left and right hand sides, with associated proof rules. Unfortunately, they use program schemas as the representation of the standard forms which has the difficulties discussed above.

Another formalism some have found attractive for codifying programming knowledge is formal grammars. For example, Ruth [30] constructed a grammar (with global switches to control conditional expansions) Which represented the class of programs expected to be handed as exercises in an introductory PL/1 programming class. This grammar was used in a combination of top-down, bottom-up and heuristic parsing techniques in order to recognize correct and near-correct programs. Miller and Goldstein [22] also used a grammar formalism (implemented as an augmented transition network) to represent classes of programs in a domain of graphical programming with stick figures. The major shortcoming of these grammars from the point of view of the programmer's apprentice is their lack of a clear semantics upon which a verification methodology can be based.

3. A Situational Calculus

Before defining the plan calculus more formally, we first need to introduce the underlying logical foundations using a *situations! calculus* similar to the one developed by Green [13] and used by McCarthy and Mayes [20]. A situational calculus is a variant of predicate calculus in which certain variables and constants are interpreted as denoting *situations.* Situations can be thought of informally as "instants in time" or "states of the workf.

1. As Opposed to the folding-unfolding and similar transformations of Burstall and Darlington [5] which are intended to be a small set of very general transformations which are formally adequate, but which must be composed appropriately to construct intuitively meaningful implementation steps.

Side Effects

One basic issue to be addressed in the logical foundations of the plan calculus is the description of *mutable objects.* In programming, as in the everyday world of physical objects, objects can change their behavior (or properties) over time without changing their identity. The approach taken by Green to this problem was to add *an* extra situational variable to the various function and relation symbols which described the time-dependent aspects of objects. So for example, for a mutable set A he would write member(x,A,s) to assert that x is a member of A at time s. At some other time t=s, it then might be the case that ¬member(x,A,t). We then say that A has been *modified* or that a *side effect* has occurred.

This situational notation becomes awkward, however, when one introduces defined relationships between objects. For example, suppose we wish to assert that between situations s and t some elements may have been removed from set A, but none have been added. The appropriate relation to use here is subset. In Green's approach we *are* forced to define subset as follows, adding two situational arguments

subset(A,B,s,t) == Vx [member(x,A,s) *D* member(x,6,t)].

We then would then assert subset(A,A,t,s) to specify the indicated side effect.

This paper proposes *ar* alternative situational calculus which allows us to preserve the standard algebra of set relations. Mutable objects are treated formally here as functions from situations to *behaviors.* Behaviors are mathematical domains, such as sets, in which there are no side effects, i.e. in which an axiom of extensionality similar to the one for sets applies ' A mutable set, such as A, is then formally a function from situations to sets. So for example we could write

A(s)={1,2,3}
A(t)-{1,2} where s≠t.

and A(t)£A(s) to specify the side effect discussed above.

This is the most straightforward example of specifying side effects. Additional complexity enters in when side effects are combined with hierarchical objects (objects with objects as parts) and multiple points of view. The representation of such side effects in the situational calculus in these more complex cases will be touched upon slightly in the upcoming sections on data plans and overlays. These topics are treated in detail in [28}, Shrobe [34] also discusses some techniques for reasoning about such complex side effects

Control Flow

In addition to distinguishing the different behavior states of mutable objects, situations can be used to represent control information. This is achieved formally by introducing a'primitive partial order on situations, called Precedes Intuitively, this relation captures the notion of states occuring before or after other states. This relation also makes it possible to talk about cyclic computations in which all objects return to the same state as at some earlier time.

Another basic feature of control flow we need to deal with in the situational calculus is conditionals. To do so formally, we Introduce a distinguished constant J_ into the domain of situations, such that Vs precedes(s.x)

Intuitively, _ represents a situation which is never reached. As we shall see in the following section, j_ appears in the axioms for tests as a way of saying that the two branches of a test *are* mutually exclusive. X also gives us the power to talk about the termination of a loop.

Computations

Given objects and situations, we can by induction construct a domain of n-tuples (and n-tuples of n-tuples) in which the base components are objects and situations. We call each such n-tuple

1. The axiom of extensionality for sets states that two sets are equal if and only if they have exactly the same members.

a *computation.* So for example, the 3-tuple <A,s,t> is a computation involving the object A and situations s and t. If the assertions given in the example above for A, s and t hold, then this computation involves a side effect to the mutable set A.

As we shall see in the next section, a plan is formally a *predicate* on computations (with the syntactic variation of referring to the components of an n-tuple by selector functions rather than by numerical index).

4 Plans

We *are* now in a position to give a formal definition of the plan calculus. In doing so, it is important to distinguish three levels of definition:

(i) *PLan Diagrams.*

This is the diagrammatic notation illustrated in Fig. 1 and Fig. 2. Historically this was the first level of the plan calculus to be developed and is still the notation which is used most by the author for intuition and explanation. It is also the abstract "mental language" of the programmer's apprentice, i.e. the most natural language in which to describe its operations and strategies (There of course also has to be a concrete implementation of this language in computer memory, as discussed below)

(ii) *Formal Semantics.*

A systematic method is given to translate each form of plan diagram into the axiomitization of a predicate in the situational calculus. This axiomatization provides the rules of inference on plans, veriflability and additivity (combination of plans has the same formal properties as union of axiom systems) Note also that for presentation purposes the actual formulae associated with some constraints in plans will often be omitted from plan diagrams, and the existence of a constraint just indicated by an arc between the constrained parts.

(ii) *Implementation.*

Several different representations of the plan calculus as Lisp data structures have been implemented and used by the author, Shrobe and Waters. Two early versions of the implementation used a general purpose fully inverted assertional data base with pattern matching (similar to the one used in Conniver [21]) in which a separate assertion was stored to represent each feature (box, arrow, etc.) of a plan diagram A subsequent re-implementation by Waters optimized this data base by providing specific access paths tailored to the specific assertion types and pattern matching that was required by the current programmer's apprentice system. Most recently a third version was implemented using McAllester's truth maintenance system [19] in which the logical axioms are represented explicitly with additional extra-logical annotation to encode the diagram level information The implementation level will not be discussed further in this paper, other than to show the external Lisp form in which plan definitions are entered in the most recent version

The basic idea of a plan, as used in the programmer's apprentice, comes from *an* analogy between programming and other engineering activities [26]. "Plans" of various Kinds are used by many different kinds of engineers. For example, an electrical engineer uses circuit diagrams and block diagrams at various levels of abstraction; a structural engineer uses large scale and detailed blue prints which show both the architectural framework of a building and also various subsystems such as heating, wiring and plumbing; a mechanical engineer uses overlapping hierarchical descriptions of the interconnections between mechanical parts and assemblies

Programming is viewed here as a process involving the construction and manipulation of specifications at various levels of abstraction. In this view, there is no fundamental distinction between specifications and programs. A program (eg in lisp) is merely a specification which is detailed enough to be carried out by some particular interpreter. This view is consistent with the current trend in computer science towards wide spectrum languages. The advantage of this approach is that various parts of a program design can be refined to different degrees without

Intervening shifts of formalism

The current plan calculus is based on a very simple model of computation (some limitations of this model and possible future extensions are discussed in the conclusions section). In this model all computations are composed out of three types of primitives: operations, tests, and primitive data objects. Corresponding to each of these primitive types, there is a primitive specification form in the plan calculus. Operations are specified by input-output specifications (preconditions and postconditions). Tests are specified by a condition on the inputs which determines whether the test succeeds or fails Primitive-data objects are specified an appropriate mathematical theory, such as numbers, sets or functions.

Plans are composite specifications constructed by the uniform mechanism of defining parts (called *roles)* and *constraints.* Two kinds of plans are distinguished according to the types of the roles. *Data plans* specify data structures whose roles are primitive data objects or other data structures Data plans Thus embody a kind of data abstraction.

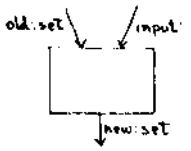
Temporal plans specify computations whose parts are operations, tests, data structures or other composite computations. The plan for a search loop in Fig 1 is an example of a temporal plan. In addition to arbitrary logical constraints between roles, temporal plans also include data flow and control flow constraints. Temporal plans thus embody a kind of control abstraction Since temporal plans can have embedded data plans, and the same compositional mechanism is used for both, control and data abstraction are unified in the plan calculus. The following sections describe the diagrams and formal semantics for each kind of plan in detail.

Input-Output Specifications

In plan diagrams an input-output specification is drawn as a solid rectangular box with solid arrows entering at the top and leaving the bottom, as shown in Fig. 3. Each arrow entering at the top represents an input; each arrow leaving the bottom represents an output. Constraints between inputs (preconditions) and between outputs and inputs (postconditions) are usually omitted in plan diagrams.

The definition of Set-add in the external format of the current data base implementation is shown in Fig. 3 next to the diagram. Set-add has two inputs Old (constrained to be a set) and Input (which may be an object of any type); and no other preconditions. The only output of Set-add is named New (a set). The postconditions of Set-add specify that the New set has exactly the same members as the Old set, with the sole addition of the Input. In terms .of its semantics, this input-output specification is translated into a type predicate on computations, Set-add, defined as shown below.[1]

Figure 3. Input-output Specification for Adding to a Set.




```
(DEFIOSPEC SET-ADD
   (((OLD SET) INPUT)                ;inputs
   ((NEW SET))                       ;outputs
   ())                               ;preconditions
   ((MEMBER INPUT (NEW))             ;postconditions
    (FORALL X (IMPLIES (MEMBER X (OLD))
                       (MEMBER X (NEW))))
    (FORALL X (IMPLIES (AND (MEMBER X (NEW))
                            (NOT (= X INPUT)))
                       (MEMBER X (OLD))))))
```

1. The axioms in this paper are slightly simplified by ignoring the fact that certain functions are partial. An axiom of extensionality for each plan type (i.e. equality of instances follows from equality of all the parts) is also being omitted here. The complete axioms are given in [28].

Set-add(α) ≡

```
[ precedes(in(α),out(α)) ∧ co-occur(in(α),out(α))
∧ set(old(αXin(α))) ∧ set(new(αXout(α)))
∧ member(input,new(αXout(α)))
∧ ∀x [ member(x,old(αXin(α)))
        ⊃ member(x,new(αXout(α))) ]
∧ ∀x [ member(x,new(αXout(α))) ∧ x≠input
        ⊃ member(x,old(αXin(α))) ] ]
```

Note that each input and output name becomes a function symbol which is the selector function for a component of a computation. Such functions are called *role functions* Thus input-output specifications (and also tests), although introduced as primitive above, are in fact composite from the standpoint of the formal semantics. Input and output names are treated the same as role names in other composite plans.

Note two role functions in the definition above, In and Out, were implicit in the DEFIOSPEC form. These are situational roles which correspond intutively to the situations immediately preceding and immediately following the execution of the specified operation. The constraint between them is that the In situation precedes the Out situation, and that they co-occur. Co-occur is an equivalence on situations which is defined as follows

Co-occur(s,t) ≡ [s=⊥ ↔ t=⊥]

This means that if the In situation is reached, it follows that the Out situation is reached, i.e. the operation terminates; and conversely, if the In situation does not occur (i.e. S«JL), then the Out situation does not occur. This converse implication, together with the axiomitizalion of tests shown in the next section, guarantees that none of the situations which follow in control flow from the failure side of a test occur

Finally, note the terms old(aXm(a)) and new(aXout(a)) in the definition of Set-add. Input and output role functions return *objects* which must be applied to the appropriate situational arguments to talk about their behavior.. Thus Old and New are mutable sets; however this specification makes no commitment as to whether or not the Input object is added by side effect (Note that we don't care about the behavior of the Input object here, only its identity.) Whether or not a side effect is allowed depends on the larger plan in which a particular use of Set-add appears, specifically on whether the Old set is used again after the operation. Formally, the specification of the side effect comes down to the question of equality between the Old *and* New objects So for example, a specialized form (the specialization mechanism used here will be explained further below) of Set-add can be defined by adding one more simple ppstcondition which stipulates that the addition be achieved by side effect.

```
(DEFIOSPEC IMPURE-5ET-A00 SPECIALIZATION SET-AOO
   (010 INPUT)
   (MEW)
   ()
   ((• 010 NEW)))
```

The meaning of this definition is shown oelow. Note that whether or not the role function is to be applied to a situational argument is encoded in the data base input notation by the presence or absence of parentheses around the role name in the constraint expression.

Impure-set-add(α) ≡ [set-add(α) ∧ old(α)≠new(α)]

Test Specifications

In plan diagrams a test specification is drawn as a solid rectangular box with a divided bottom part, as shown in Fig. 4. The inputs and outputs of a test specification and their types are indicated in the same way as the inputs and outputs of an input-output specification. A test also has preconditions and postconditions, just like *an* input-output specification. A test specification differs from an input-output specification in that two distinct output situations, named Succeed and Fail, *are* implicitly specified. Which one occurs depends on whether or not a given relation (called the *condition* of the test) holds true between the Inputs. Control flow arcs originating from either the part of the test box marked S (for succeed) or the part marked F (for fail) *are* then used to indicate which other parts of a plan are executed depending on the test.

The definition of the test specification Apply-predicate written in the format of the current data base implementation is also shown in Fig 4. Note that this is the test specification used in the Found role of the Search-loop plan of Fig. 1. The two exit roles of Fig. 1 illustrate *joins* which are the mirror images of tests in the plan calculus. Joins are required in conditional plans to specify what the output is in each case

Apply-predicate has two inputs Criterion (a predicate) and Input (any object to which the predicate is applicable); and no outputs. The test succeeds when the Criterion is true of the Input; otherwise it fails. In terms of its semantics, this test specification is translated into a type predicate on computations with three situational roles, In, Succeed and Fail, as shown below.

Apply-predicate(α) ≡

```
[ precedes(in(α),succeed(α)) ∧ precedes(in(α),fail(α))
∧ mutex(succeed(α),fail(α))
∧ predicate(criterion(α))
∧ member(input(α),domain(criterion(αXin(α))))
∧ [ apply(criterion(αXin(α)),input(α))
        ⊃ co-occur(in(α),succeed(α)) ]
∧ [ ¬apply(criterion(αXin(α)),input(α))
        ⊃ co-occur(in(α),fail(α)) ] ]
```

Mutex(s,t) ≡ [s=⊥ ∨ t=⊥]

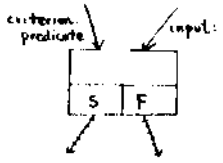
From this way of defining tests it is possible to reason both backward and forward in time. For example, if the In situation is reached (i.e. $in(\alpha) \neq \perp$) and the Criterion is true, then it follows that the Succeed situation is reached. Conversely, if we know that the either the Succeed or Fail situation is reached, it follows that the In situation must have been reached.

Data Plana

Data plans are plans whose roles are primitive data objects or other data plans. In plan diagrams primitive data objects are drawn as solid ovals; data plans are drawn as dashed ovals. Data plans *are* used to represent standard data structure configurations which may be used in the implementation of more abstract data types. For example, the data plan Indexed-Sequence, shown in Fig. 5, represents the common cliche of a sequence (typically implemented more concretely as an array) with an associated index pointer. This configuration is typically used to implement such things as buffers, queues, and stacks (Implementation relationships *are* represented using overlays, which will be discussed further in an uDcoming section.)

The plan Indexed-sequence has two roles; Base (a sequence)

Figure 4. Test Specification for Applying a Predicate.




```
(DEFTEST APPLY-PREDICATE
   ((CRITERION PREDICATE) INPUT)          ;inputs
   ()                                     ;outputs
   ((MEMBER INPUT (DOMAIN (CRITERION))))  ;preconditions
   ()                                     ;postconditions
   (APPLY (CRITERION) INPUT))             ;test condition
```

1. More complicated tests with more than two cases can be represented by composing binary tests. Alternatively, the test notation may be generalized to more than two cases

•nd Index (an integer); and the constraint that the Index is between zero and the length of the sequence. The definition of Indexed-sequence in the external format of the data base is also shown in Fig. 5.

Data plans as a mechanism for data abstraction are similar in certain ways to to algebraic axioms [14,12] and abstract data types [17]. The constraints between roles of a data plan are similar to what is called the *invariant* in other formalisms. From this point of view, the semantic translation of a data plan can be thought of as a data type predicate. For example, Indexed-sequence, is defined as shown below.

```
Indexed-sequence(δ) ≡
        ∀s [ sequence(base(δ)(s)) ∧ integer(index(δ)(s))
                ∧ gt(index(δ)(s),0)
                ∧ le(index(δ)(s),length(base(δ)(s))) ]
```

Algebraic axioms and abstract data types, however, do not allow One to specify side effects on data structures -- all operations *are* purely functional. In the plan calculus, we single out the primitive selector operations of the data abstraction, which become the role functions. Side effects are then specified in terms of changes to the roles of a data structure. Note that Indexed-sequence is defined as a predicate on behaviors not objects, i.e. *I* above is meant to denote a 2-tuple with components named Base •nd Index. A mutable indexed sequence is an object D such that, for example, D(s)-$. Aspects of this formalization of mutable data structures are similar to the approaches taken by Earley [9], by Reynolds [24] for reasoning about arrays, and by Guttag and Horning [15]

The plan calculus also represents sharing (i.e aliasing) in composite data objects, which is not the case with algebraic axioms or abstract data types. Space does not allow a full treatment of this topic here (see [28]); the basic idea is that the parts of a composite data object can themselves be data objects, eg. base(D(s)) above is a mutable sequence In this formalization a side effect to base(D(s)) logically propagates to D(s)

Temporal Plans

Temporal plans are the most general form of plan, in which both data and control abstraction can be expressed The roles of • temporal plan may be either primitive data objects, data plans, Input-output specifications or test specifications The most typical constraints between input-output and test specification roles *are* data flow and control flow constraints An example of the diagram for a temporal plan is shown in Fig. 6 (note that Fig. 1 earlier also illustrates a temporal plan).

The Bump+update plan in Fig 6 has four roles: Bump, a subtract one operation; Update, an operation to change one term in a sequence; and Old and New, which are indexed sequence data structures. This plan captures the cliched pattern of operations on •n indexed sequence in which the index is decremented and a new term is stored. (Among other things, this is the implementation of ■ Push operation if the Old and New indexed sequences are viewed •s stacks — see section on overlays).

Solid arrows in a temporal plan diagram, such as between the Output of the Bump step and the Index input of the Update step, denote data flow, i.e. the Output of Bump becomes the Index input of Update. This notion of data flow is intuitively one of the main sources of additivity in the plan calculus. An arbitrary amount of other computation may be added between Bump and Update as long as the stipulated data flow is not disturbed. As we shall see below, data flow constraints are formalized logically as equalities between terms which denote the respective input and output ports. The additional solid lines in Fig. 6 denote equality constraints between parts other than the input and output ports of input-output specifications. For example, the Base of the Old indexed sequence becomes the Old sequence input to the Update step; the Base of the New indexed sequence is the New sequence output of the Update step.

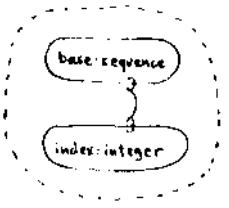
A second feature of temporal plan diagrams introduced in Fig. 6 is control flow (cross-hatched) arrows. The control flow arrow between Bump and Update means that the termination of the Bump step precedes (and implies) the initiation of the Update step. The external plan definition form for Bump4Update given in Fig. 6 summarizes all of these features. Also shown below are the input-output specifications of the Subtract-one and Newterm, which are straightforward.

```
(DEFIOSPEC SUBTRACT-ONE
   ((INPUT INTEGER))
   ((OUTPUT INTEGER))
   ()
   ((= (OUTPUT)(- (INPUT) 1))))

(DEFIOSPEC NEWTERM
   ((OLD SEQUENCE)(INDEX INTEGER) INPUT)
   ((NEW SEQUENCE))
   ((GT (INDEX) 0)(LE (INDEX)(LENGTH (OLD))))
   ((= (LENGTH (NEW))(LENGTH (OLD)))
    (= (TERM (NEW)(INDEX)) INPUT)
    (FORALL I (IMPLIES (NOT (= I (INDEX)))
                       (= (TERM (OLD) I)(TERM (NEW) I))))))
```

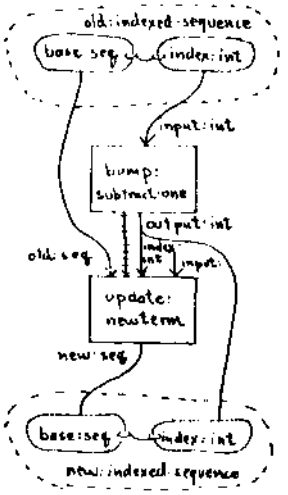
There are two further features of the Bump+update plan to note before looking at the semantic translation. First, like Set-add

Figure 5. Data Plan for an Indexed Sequance.




```
(DEFDATAPLAN INDEXED-SEQUENCE
   ((BASE SEQUENCE) (INDEX INTEGER)) ;roles
   ((GT (INDEX) 0)                   ;constraints
    (LE (INDEX) (LENGTH (BASE)))))
```

Figure 6. Temporal Plan for Updating an Indexed Sequence.




```
(DEFPLAN BUMP+UPDATE
   ;roles
   ((OLD INDEXED-SEQUENCE)(NEW INDEXED-SEQUENCE)
    (BUMP SUBTRACT-ONE)(UPDATE NEWTERM))
   ;constraints
   ((CFLOW (OUT (BUMP)) (IN (UPDATE)))
    (= ((INDEX (OLD))) ((INPUT (BUMP))))
    (= ((BASE (OLD))) ((OLD (UPDATE))))
    (= ((OUTPUT (BUMP))) ((INDEX (UPDATE))))
    (= ((OUTPUT (UPDATE))) ((BASE (NEW))))
    (= ((OUTPUT (BUMP))) ((INDEX (NEW))))))
```

this plan makes no commitment to whether or not a side effect is involved. A specialized version of Bump+update can be specified In which the Update step is specialized to an impure Newterm specification and in which the Old and New indexed sequences are constrained to be identical. Also note that the Input to Newterm (the value of the new term) is not constrained in this plan. Temporal plans are typically combined by providing data How from •nd to such ports

Shown below is the semantics of the temporal plan Bump* update which is defined, similar to input-output and test specifications, as a predicate on computations

```
Bump+update(α) ≡
      [subtract-one(bump(α)) ∧ newterm(update(α))
      ∧ cflow(out(bump(α)),in(update(α)))
      ∧ output(bump(α))(out(bump(α)))
         =index(update(α))(in(update(α)))
      ∧ ∃s [ indexed-sequence(old(α)(s))
        ∧ index(old(α)(s))(s)=input(bump(α))(in(bump(α)))
        ∧ base(old(α)(s))(s)=old(update(α))(in(update(α))) ]
      ∧ ∃t [ indexed-sequence(new(α)(t))
        ∧ index(new(α)(t))(t)=output(bump(α))(out(bump(α)))
        ∧ base(new(α)(t))(t)=output(update(α))(out(update(α)))]

Cflow(s,t) ≡ [ precedes(s,t) ∧ co-occur(s,t) ]
```

The interpretation of this formal definition is left largely to the reader. The major complexity in translating temporal plans into their logical equivalent is in providing the appropriate situational arguments. In particular, for roles which are data plans, such as Old *and* New above, we posit the existence of some situation (eg s *and t* above) in which the constraints of the data plan hold, and then assert constraints between the behavior of the parts in that situation and the input and output ports to which they are connected in the plan diagram. The default situational arguments for input and output ports here are assigned the same way as in the corresponding input-output specifications, namely In for input objects and Out for output objects.

The Bump+update plan does not illustrate an additional important feature of temporal plans, namely the use of recursion to represent loops. Recent work on Lisp interpreters [35] and compilers [36] suggests that the distinction between loops and singly recursive programs in which the recursive call is the last step of the program (so called "tail recursions") should be considered only a superficial syntactic variation. The plan calculus takes this point of view. Recursion in plan diagrams (data plans may also be recursively defined) is indicated by a spiral line as shown in Fig. 1, the plan for a search loop, introduced earlier

Programming Languages

The plan calculus makes it possible to build a programmer's apprentice which is concerned with the syntactic details of different programming languages only at its most superficial interface. In order to translate back *and* forth between a given programming language and the plan calculus, the primitives of the programming language *are* divided into two categories:

(i) The primitive *actions* and *tests* of the language, such as CAR, COR, CONS, NULL and EQ in Lisp, are represented as input-output specifications and test specifications.

(ii) The primitive *connectives,* such as PROG, CONO, StTO, GO and RETURN in Lisp, are represented as patterns of control and data flow constraints between operations and tests.

The translation from standard program text to an equivalent plan representation has been implemented for reasonable subsets Of Lisp [25], Fortran [39] and Cobol [10]. The translation from suitably restricted plans to Lisp code has also been implemented by Waters [40].

5. Relations Between Plant

Fig. 7 is a small excerpt from the current plan library which illustrates the the taxonomic structure of the library. A first observation to make about the nodes of the library shown in this example is that they are a mixture of both data abstractions (e.g. set, predicate, directed graph) and control abstractions (Find and Thread-find are input-output specifications, and in general there *are* temporal plans also).

The four types of relations which relate nodes of the library are named in the upper right hand corner of the figure next to an example of the type of line or arrow used to represent each relation In the body of the figure. The first relation, *component,* simply indexes which plans are used in the definition of other plans, as for example Indexed-sequence was used in the definilion of Bump+update above. The next two relations are simple inheritance relationships by which a plan inherits the roles and constraints of another plan and then adds its own additional roles or constraints or both. *Specialization* is the relation in which only constraints *are* added; *extension* involves adding roles (and usually, constraints between the new roles and the inherited ones).

Overlays are the most important type of relation between plans in the library because overlays represent significant additional knowledge, such as the details of how one abstraction may be Implemented in terms of another. The diagram notation and semantics for overlays are explained in the following section.

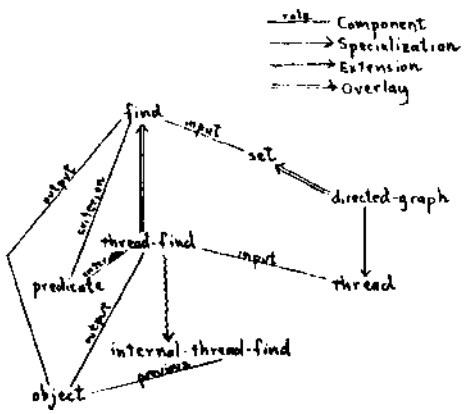
Starting with Set in Fig 7, a typical question we might want to ask in the course of *program* analysis or synthesis is: what plans <e.g. input-output specifications) are there in the library which use sets? This type of question is answered in general by looking up the composition relation; in this example from Set to Find Find is an input-output specification with three roles an Input set, a Criterion predicate and an Output object; its precondition is that there exist an element of the Input set which satisfies the Criterion; its postcondition is that the Output is such an element.

Overlays pointing to a node answer questions of the form: what implementations *are* available for Find operations? In this example, one answer is that Find may be implemented as Thread-find, where the Input set is implemented as a thread. (Note that Thread is a specialization of Directed-graph in which each node has at most one predecessor or successor and there are no cycles.) The input-output specifications of Thread-find are shown below. It has three roles similar to Find: an Input thread, a Criterion predicate and an Output object; its precondition is that there exist a node of the Input thread which satisfies the Criterion; its postcondition is that the Output is such a node.

```
(DEFIOSPEC THREAD-FIND
   ((INPUT THREAD)(CRITERION PREDICATE))
   (OUTPUT)
   ((EXISTS X (AND (NODE (INPUT) X)
                   (APPLY (CRITERION) X))))
   ((NODE (INPUT) OUTPUT) (APPLY (CRITERION) OUTPUT)))
```

Finally, extension is a relation between plans which can be used to access useful variations of a plan. For example, in the figure an extension of Thread-find is Internal-thread-find. The Input-output specifications of Internal-thread-find are shown below.

Figure 7. Excerpt from Plan Library Illustrating Taxonomic Relations



```
(DEFIOSPEC INTERNAL THREAD-FIND EXTENSION THREAD-FIND
   (INPUT CRITERION)
   (OUTPUT PREVIOUS)
   ((NOT (APPLY (CRITERION)(ROOT (INPUT))))))
   ((SUCCESSOR (INPUT) PREVIOUS OUTPUT)))
```

Note that an additional output role, Previous, has been added with the constraint that it be the predecessor of the Output node In the Input thread. This is a very common Kind of operation on directed graphs when splicing of. nodes in and out is being performed. In order to be used properly, Internal-thread-find must also have the additional precondition (from which its name is suggested) that the Criterion is false for the root node of the Input thread.

Overlays

Overlays provide a mechanism »n the plan calculus for representing the relationship between two different points of view, each of which is represented by a plan. Overlays are similar to Sussman's "slices" [37], which he uses to represent equivalences in electronic circuit analysis and synthesis. Formally, an overlay is a mapping from instances of one plan to instances of another. Such mappings are a very general mechanism; the intuitive import of various overlays depends on what kind of plans are involved, whether the mapping is many-to-one or one-to-one, and how the mapping is defined.

An example of an overlay between a temporal plan and an input-output specification is shown in Fig. 8. Intuitively, this overlay expresses how a Push operation can be implemented by the Bump+update plan discussed above, given that the stack is implemented as an indexed sequence. Note that the diagram for an overlay is made up of a plan on the left hand side (which is the domain of the mapping), a plan on the right hand side (which is the range of the mapping), and a set of hooked lines showing a set of *correspondences* between roles of the two plans (which defines the mapping). In the external format of the current data base, an overlay is defined as follows

```
(DEFOVERLAY BUMP+UPDATE>PUSH (BUMP+UPDATE PUSH)
   (= (OLD PUSH) (INDEXED-SEQUENCE>STACK (OLD BUMP+UPDATE)))
   (= (NEW PUSH) (INDEXED-SEQUENCE>STACK (NEW BUMP+UPDATE)))
   (= (INPUT PUSH) (INPUT (UPDATE BUMP+UPDATE)))
   (= (IN PUSH) (IN (BUMP BUMP+UPDATE)))
   (= (OUT PUSH) (OUT (UPDATE BUMP+UPDATE))))
```

Notice that each overlay has a name, eg Bump-fupdate>push,[1] which will become a function symbol in the formal semantics. Correspondences between roles are either simple equalities, as in

(= INPUT PUSH) (INPUT (UPDATE BUMP*UPDATE))) ,

which says that the Input to Push (the object which becomes the head of the new stack) corresponds to the Input of the Update step in the Bump+update plan; or correspondences which defined in terms of other overlays, as in

(= (OLD PUSH) (INDIXED-SEQUENCED STACK (OLD BUMP+UPDATE)))

The overlay Indexed-sequence>stack is an example of a many-to-one overlay between data plans. The basic idea of this overlay is that an indexed sequence may be viewed as the implementation of a stack In which the head is the term of the sequence indexed by the current index, and the tail is, recursively, the sequence with index one greater. The formal definition of this overlay will not be shown here, as it is very similar to what is called an abstraction function in abstract data types The formal semantics Of the Bump+update>push overlay is the following function definition.

β=Bump+update>push(α) ≡
[old(β)=indexed-sequence>stack(old(α))
∧ new(β)=indexed-sequence>stack(new(α))
∧ input(β)=input(update(α))
∧ in(β)=in(bump(α))
∧ out(β)=out(update(α))]

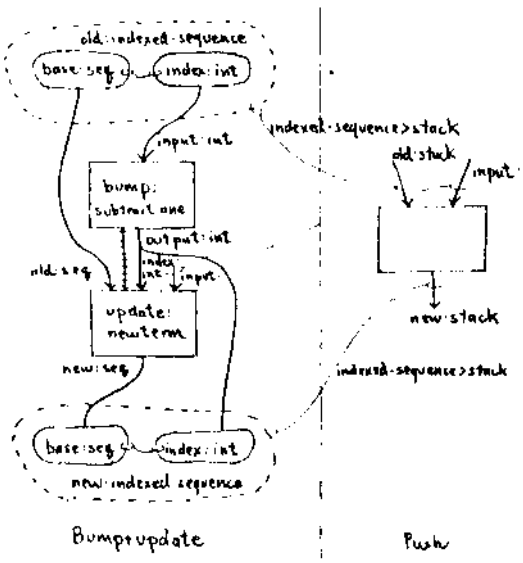
1. This symbol is intended to be read "Bump and Update as Push".

Implementation overlays such as Bump+update >push and Indcxed~sequence>stack are typically many-to-one. There are also one-to-one overlays in the library, which are naturally thought of as providing alternative points of view or transformations of data or control structure. For example, a Lisp list can be viewed alternatively as a stack (i.e. a recursively defined data structure), as a labelled directed graph (in particular a thread) wherein the nodes are Cons cells connected by the Cdr relation and labelled by the Car relation, or as a sequence wherein the i-th term is the Car of (i-l)th Cdr. Each of these data plans provides its own vocabulary for specifying properties of a data structure and a standard set of manipulations on it. There are one-to-one overlays in the library between them so that the appropriate point of view can be used depending on the properties and manipulations of interest, There *are* also one-to-one overlays in the library between temporal plans, such the Overlay introduced earlier in Fig. 2 which captures the relationship in general between singly recursive programs which accumulate "on the way down" (eg reverse copying a list) and those that accumulate "on the way up" (eg. copying a list).

Other overlays of particular interest are those which map from temporal plans to data plans, in particular from recursive temporal plans to lists. These latter are called *stream* overlays and embody a kind of abstraction in which the objects filling a given role at all levels in a recursive plan are viewed as a single data structure. For example, there is a stream overlay in the library by means of which the succession of Input's to the Found test in the Search-loop plan are viewed as a list; and if that overlay is composed with the overlay to view a list as the implementation of a set, then Search-ioop can be viewed as the implementation of the Find specification discussed above This method of abstraction provides a very powerful means of decomposing loops and recursions in a way which makes their behavior much clearer than the standard invariant assertion method For a further discussion of this idea see Waters [39].

A final point to note regarding the use of overlays is that they are not only part of the laxonomic structure of the plan library, they are also used to construct the refinement tree of a particular program. This tree encodes the design history of a program at many levels of detail. A way in which overlays differ from other refinement tree notations is that overlays permit *Overlapping* implementations [29], where components that a*re* distinct at one level of abstraction share parts at lower levels. This is a real

Figure 8. Implementation Overlay for Push on Indexed Sequence.



feature of good engineering design and is an important way in which overlays are an improvement over previous notations.

6. Final Remarks

The emphasis in this paper on representation rather than reasoning has allowed no mention of an important dimension of the plan calculus In the programmer's apprentice, reasoning about plans takes place within a system [19] which maintains explicit dependencies between each assertion in the data base. These dependencies are crucial for supporting evolutionary design and debugging of programs. Each overlay in the library has a pre-compiled network of dependencies between the specifications on the left and right hand sides (called its *Ideological* structure) which is instantiated along with the parts, constraints and correspondences when it is used

There are several areas in which we have found the current plan calculus deficient *and* hope to extend it further in the future Important among these are the following:

(i) *Performance Considerations* A way of formalizing time and space trade-offs is needed, both for the library and for reasoning about particular programs.

(ii) *Error Handling.* The non-local flow of control typically Involved in error handling code is not well represented in the current plan calculus.

(iii) *Global Shared Data.* Programs which communicate primarily by inserting *and* retrieving information in a global shared data base *are* not well decomposed by the data flow ideas in the current calculus.

Finally, we intend to investigate the usefulness of the plan calculus in other planning contexts. In particular, I believe there is potential for a fruitful flow of ideas back and forth between program understanding (analysis, synthesis and verification) and robot planning (e.g. Abstrips [31]). Many of the insights underlying the plan calculus originate from work in robot planning;

1 also believe some of the techniques developed in the plan calculus for representing and reasoning about overlapping data structures and side effects have bearing on aspects of the classical "frame problem" [20] in robot planning.

Bibliography

[I] P BALZER, "TRANSFORMATIONAL IMPLEMENTATION AN EXAMPLE", *IEEE TRANS. ON SOFTWARE ENC,* VOL 7, NO I, JANUARY, 1981

(2) DR BARSTOW, "AUTOMATIC CONSTRUCTION OF ALGORITHMS AND DATA STRUCTURES USING A KNOWLEDGE BASE OF PROGRAMMING RULES", STANFORD AIM-308, NOV 1977

(3) S BASU AND J MISRA, "SOME CLASSES OF NATURALLY PROVABLE PROGRAMS", *2ND INT. CONf. ON SOFTWARE ENC,* SAN FRANCISCO, CAL, OCT. 1976

(4) M BROY AND P PEPPER, "PROGRAM DEVELOPMENT AS A FORMAL ACTIVITY", *IEEE TRANS. ON SOFTWARE ENC.,* VOL 7, NO I, JANUARY, 1980

(5) RM BURSTALL ANO JI DARLINGTON, "A TRANSFORMATION SYSTEM FOR DEVELOPING RECURSIVE PROGRAMS", J *OF THE ACM,* VOL 24, NO I, JANUARY, 1977

(6) TE CHEATHAM, "PROGRAM REFINEMENT BY TRANSFORMATION", *5TH INT CONF. ON SOFTWARE ENC,* SAN DIEGO, CAL , MARCH, 1981

(7) 0 J DAHL ANO K NYGAARD "SIMULA - AN ALGOL-BASED SIMULATION LANGUAGE", *COMM. OF THE ACM,* VOL 9, NO 9, SEPTEMBER 1966, *PP* 671-678

(8) JB DENNIS. "FIRST VERSION OF A DATA FLOW PROCEDURE LANGUAGE". *PROC OF SYMPOSIUM ON PROGRAMMING,* INSTITUT DE PROGRAMMAT0N, U OF PARIS, APRIL 1974, *PP* 241-271

(9) J EARLEY. "TOWARD AN UNDERSTANDING OF DATA STRUCTURES", *COMM OF THE ACM,* VOL 14, NO 10, OCTOBER 1971, PP 617-627

(10) G FAUST, "SEMIAUTOMATIC TRANSLATION OF COBOL INTO HIB0L", (MS THESIS), MIT/ICS/TR-256, MARCH, 1981

(I I] SL GERHART, "KNOWLEDGE ABOUT PROGRAMS A MODEL ANO CASE STUDY", IN *PROC. OF INT CONF. ON RELIABLE SOFTWARE,* JUNE 1975, PP 84-95

(12) J A GOGUEN, JW THATCHER, ANO EG WAGNER, "AN INITIAL ALGEBRA APPROACH TO THE SPECIFICATION, CORRECTNESS, ANO IMPLEMENTATION OF ABSTRACT OAT A TYPES," *CURRENT TRENDS IN PROGRAMMING METHODOLOGY, VOL. IV,* (ED RAYMOND YEH), PRENTICE-HALL, 1978

(13) C GREEN. "THEOREM PROVING BY RESOLUTION AS A BASIS FOR OUtSTION-ANSWERING SYSTEMS, *MACHINE INTELLIGENCE 4,* D MICHIE AND B MEIUER, EDS, EDINBURGH UNIVERSITY PRESS, EDINBURGH, SCOTLAND, 1969

(14) J GUT TAG. "ABSTRACT DATA TYPES AND THE DEVELOPMENT OF DATA STRUCTURES", *COMM OF THE ACM,* VOL 20, NO 6, JUNE 1977, PP 396-404

(15) J GUTTAG AND J J HORNING. "FORMAL SPECIFICATION AS A DESIGN TOOL", *7TH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES,* LAS VEGAS, JANUARY, 1980, PP 251 261

(16) B LISKOV *ET. AL.,* "ABSTRACTION MECHANISMS IN CLU", *COMM. OF THE ACM,* VOL 20, NO 8, AUGUST 1977, PP 564-576

(I7] BM LISKOV ANO 5N ZILLES, "AN INTRODUCTION TO FORMAL SPECIFICATIONS OF DATA ABSTRACTIONS," *CURRENT TRENDS IN PROGRAMMING METHODOLOGY, VOL. I,* (ED RAYMOND YEH), PRENTICE-HALL, 1977

(18) Z MANNA, *MATHEMATIC THEORY OF COMPUTATION,* MCGRAW-HILL, 1974

(19) DA MCALLESTER. AN OUTLOOK ON TRUTH MAINTENANCE", MIT/AIM-551, AUGUST, 1980

(20) J MCCARTHY AND P HAYES. "SOME PHILOSOPHICAL PROBLEMS FROM THE STANDPOINT OF ARTIFICIAL INTELLIGENCE", *MACHINE INTELLIGENCE 4,* D MICHIE AND B MELTZER, EDS, EDINBURGH UNIVERSITY PRESS, EDINBURGH, SCOTLAND. 1969

(21) D MCOERMOTT ANO G J SUSSMAN, "THE CONNIVER REFERENCE MANUAL", MIT/AIM-259A, 1973

(22) ML MILLER ANO I GOLDSTEIN, "PROBLEM SOLVING GRAMMARS AS FORMAL TOOLS FOR INTELLIGENT CAI", *PROC OF THE ASSOC FOR COMPUTING MACHINERY,* 1977

(23) J MISRA, "AN APPROACH TO FORMAL DEFINITIONS AND PROOFS OF PROGRAMMING PRINCIPLES", *IEEE TRANS. ON SOFTWARE ENC,* VOL SE-4, NO 5, SEPTEMBER 1978, PP 410-413

(24) JC REYNOLDS, "REASONING ABOUT ARRAYS", *COMM Of THE ACM,* VOL 22, NO 5, MAY 1979, *PP* 290-298

(25) C RICH AND HE SHROBE, "INITIAL REPORT ON A LISP PROGRAMMER'S APPRENTICE ', (MS THESIS). MIT/AI/TR-354. DECEMBER 1976

(26) C RICH. HE SHROBE, RC WATERS, G J SUSSMAN, AND CE HEWITT, "PROGRAMMING VIEWED AS AN ENGINEERING ACTIVITY", (NSF PROPOSAL), MIT/AIM-459, JANLJARY, 1978

(27) C RICH, HE SHROBE, AND RC WATERS, "AN OVERVIEW OF THE PROGRAMMER'S APPRENTICE", *PROC. OF 6TH INT. JOINT CONf. ON ARTIFICIAL INTELLIGENCE,* TOKYO, JAPAN, AUGUST, 1979

(28) C RICH, "INSPECTION METHODS IN PROGRAMMING-. MIT/AI/TR-604, (PHD •THESIS), DECEMBER, 1980

(29) C RICH, "MULTIPLE POINTS OF VIEW IN MOOEUNG PROGRAMS", *PROC. OF WORKSHOP ON DATA ABSTRACTION, DATA BASES AND CONCEPTUAL MODELING, ACM SIGPLAN NOTICES,* VOL 16, NO I, JANLJARY, 1981, PP 177-179

(30) G RUTH, "ANALYSIS OF ALGORITHM IMPLEMENTATIONS-, MIT PROJECT MAC TECHNICAL REPORT 130. (PHD THESIS), 1973

(31) ED SACERDOTI, "PLANNING IN A HIERARCHY OF ABSTRACTION SPACES", *ARTIFICIAL INTELLIGENCE,* VOL 5, NO 2, 1974, PP 115-135

(32) JT SCHWARTZ, "ON PROGRAMMING", AN INTERIM REPORT ON THE SETI PROJECT, COURANT INSTITUTE OF MATHEMATICAL SCIENCES, NEW YORK UNIVERSITY. JUNE 1975

(33) M SHAW, W A WULF, AND RI LONDON, "ABSTRACTION AND VERIFICATION IN ALPHARD DEFINING AND SPECIFYING ITERATION AND GENERATORS", *COMM OF THE ACM,* VOL 20, NO 8, AUGUST 1977. PP 553-563

(34) HE SHROBE. "DEPENDENCY DIRECTED REASONING FOR COMPLEX PROGRAM UNDERSTANDING , (PHO THESIS), MIT/AI/TR-503, APRIL 1979

(35) GI STEELE ANO G J SUSSMAN, 'THE ART OF THE INTERPRETER, OR. THE MODULARITY COMPLEX (PARTS ZERO, ONE, AND TWO), MIT/AIM-453, MAY 1978

(36) GL STEELE, "RABBIT A COMPILER FOR SCHEME (A STUOY IN COMPILER OPTIMIZATION), MIT/AI/TR-474, MAY, 1978

(37) G J SUSSMAN, "SLICES AT THE BOUNDARY BETWEEN ANALYSIS AND SYNTHESIS", *ARTIFICIAL INTELLIGENCE AND PATTERN RECOGNITION IN COMPUTER AIDED DESIGN* LATOMBE, ED, NORTH-HOLLAND, 1978

(38) RC WATERS. "AUTOMATIC ANALYSIS OF THE LOGICAL STRUCTURE OF PROGRAMS", MIT/AI/TR-492, (PHD THESIS), DECEMBER, 1978

(39) RC WATERS, "A METHOD FOR ANALYZING LOOP PROGRAMS". *IEEE TRANS ON SOFTWARE ENG.* VOL SE-5. NO 3. MAY 1979, PP 237-247

(40) RC WATERS. "A KNOWLEDGE BASED PROGRAM EDITOR", *PROC OF 7THINT. JOINT CONF. ON ARTIFICIAL INTELLIGENCE,* VANCOUVER, CANADA, AUGUST, 1981

(41) T WINOGRAO, "BREAK THE COMPLEXITY BARRIER (AGAIN)", *PROC OF THE SIGIR-SIGPLAN INTERFACE MEETING,* NOVEMBER, 1973

(42) N WIRTH, *SYSTEMATIC PROGRAMMING, AN INTRODUCTION,* PRENTICE-HALL, 1973