MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLEGENCE LABORATORY

A.I. Memo 506 January 1979

COMPUTER AIDED EVOLUTIONARY DESIGN FOR SOFTWARE ENGINEERING

Charles Rich, Howard E. Shrobe, and Richard C. Waters

ABSTRACT - We report on a partially implemented interactive computer aided design tool for software engineering. A distinguishing characteristic of our project is its concern for the evolutionary character of software systems. Our project draws a distinction between algorithms and *systems*, centering its attention on support for the system designer. Although verification has played a large role in recent research, our perspective suggests that the complexity and evolutionary nature of software *systems* requires a number of additional techniques, which are described in this paper.

The managing of complexity is a fundamental issue in all engineering disciplines. We identify three major techniques used in mature engineering fields which seem applicable to the engineering of software systems: incremental modelling, multiple and almost hierachical decomposition, and analysis by inspection. Along these lines we have (i) Constructed a *plan library* to aid in analysis by inspection (the analysis of a program based on identifying standard algorithms and methods in it); (ii) Identified a small set of *plan building methods* which can be used to decompose a software system into loosely coupled subsystems; (iii) Developed the technique of *temporal abstraction* which makes it possible to model a program from a viewpoint which clearly separates the actions of generators and consumers of data and (iv) Developed a dependency-based reasoning system uniquely suited to incremental and evolutionary program analysis. These methods are substantially language independent and have been applied to programs written in several commonly used languages.

*This paper was adapted from a proposal to the National Science Foundataion.

This report describes research conducted at the Artificial Intelligence Laboratory and at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for the Artificial Intelligence Laboratory's Artificial Intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643. Support for the Laboratory for Computer Science's research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661.

I. The Nature of The Problem

Large software systems are expensive to design and implement, and even more expensive to maintain. The following anecdote is indicative of the kind of difficulties which are all too typical. A major commercial firm undertook the development of a large financial software system about seven years ago. The project began with the careful development of a complete design which was then implemented. This effort took four or five years, required six full time programmers and cost roughly five million dollars. During the course of the implementation effort, many of the initial design features were found to be unsatisfactory. Furthermore, the firm's business practices and the applicable government regulations underwent numerous revisions as time went by. These factors resulted in a series of modifications to the system which were documented poorly if at all. Although the program at present is known to have certain bugs, it has been very useful. In fact it is so useful that the firm would like to modify the program for use in other departments and on other computers. However, no one really knows how it works anymore. The current staff of the project has no programmer who has been involved with the system for more than eleven months. The only complete documentation is the original design, now six years out of date. The firm is faced with the prospect of redesigning and recoding the entire system from the ground up.

The evolutionary nature of systems is a central feature in the current software crisis. The specifications change, the design changes, and, as bugs are discovered, the implementation must be changed to fix them. One of the driving forces behind this is the desire for new features. This is prompted by two main factors. First, it is not possible for the designers or the potential users of a system to foresee all of the opportunities for the system's use. Second, the environment in which the system operates is itself subject to change. New regulations, business practices and technology appear and force modifications to the system.

A dominant problem in the design of large software systems is how to manage and limit the apparent complexity of the situation so that some reasonable solution can be produced. If all of the relevant constraints were considered at once in order to try to arrive at a perfect solution in the first place, the details would overwhelm human cognitive capacity. A more effective strategy is to start with a solution which is reasonably close to being correct, and then to modify it repeatedly until a solution is reached which meets the actual needs. Thus there is both an internal and an external cause for the evolutionary nature of software.

Automatic verification attacks the problem of evolution by attempting to eliminate the need for change. If a program is verified at the start, then bugs will not surface later and therefore the program will not have to be modified in order to fix them. However, automatic verification can be at most only part of the solution to the software problem because it does not attack the external sources of change such as changing government regulations. A second use for verification is in the certification of software systems. Such facilities are highly desirable but they do not eliminate the need for other types of support during the process of developing code good enough to warrant the effort of certification.

We suggest that what is needed in addition is a computer aided design tool which can help a programmer deal with program evolution from the initial design phase right through the continuing maintenance phase. We imagine a system (such as the Programmer's Apprentice [Hewitt & Smith 1975; Rich & Shrobe 1976,78; Waters 1976,78,79]) which can function as a "junior programmer" might. Such a system does not need to be able to certify the correctness of an entire software system. Rather, it must provide the designers and coders with relevant information which helps them to better understand the consequences of modifications. The system would be relatively knowlegeable, but not an expert programmer itself. It would be able to understand, explain, and reason about programs in terms familiar to a programmer. Its main virtue would be its ability to select from the vast annotation associated with a software system the small part which is relevant to whatever task the programmer is currently conducting. We propose to develop the fundamental principles behind such systems and to construct a prototype system which meets many of these desiderata.

II. An Idealized Scenario

In order to indicate the kind of system we have in mind, this section presents an imaginary scenerio showing an expert programmer interacting with a Programmer's Apprentice during the design and coding of a hashed associative retrieval system. Natural English dialog is used in the example in order to make the presentation more comprehensible. However, natural language understanding and generation are beyond the scope of the work proposed here.

PROGRAMMER: I want to make a hashed associative retrieval system for tree-structured facts. Every fact will be stored in each of the buckets hashed to by one of its leaf nodes. I am going to start by coding the insert routine. Here is the code:

```
(DEFUN INSERT (FACT) (INSERT1 FACT 1))
(DEFUN INSERT1 (FACT POSITION)
  (COND ((ATOM FACT) (BUCKET-INSERT FACT (HASH FACT POSITION TABLESIZE)))
        (T (INSERT1 (CAR FACT) (* POSITION 2))
           (INSERT1 (CDR FACT) (+ (* POSITION 2) 1)))))
```

APPRENTICE: I do not understand this code completely. I see that you are doing a tree-traversal over the fact terminated at atomic nodes, and that you are doing a BUCKET-INSERT for each of these terminal nodes. However, I do not understand what the computation involving POSITION is for, nor why HASH takes POSITION as an argument. The hashing algorithms I know about take only two arguments: a key, and a table size, and compute an index smaller than the size.

PROGRAMMER: In this system HASH takes two arguments in addition to the size argument: an atom from the fact and a number (POSITION) which uniquely identifies the position of the atom in the tree structure of the fact. Like a standard hash, it computes an index smaller than the size. POSITION makes it possible to distinguish between different occurences of the same atom in a fact.

APPRENTICE: I still do not completely understand how INSERT works. How is BUCKET-INSERT going to work?

PROGRAMMER: In this design, buckets will be ordered lists of facts. BUCKET-INSERT will fetch the bucket in the hash table indexed by its second argument and then use the standard ordered list insertion algorithm to splice its first argument into the bucket by side-effect.

APPRENTICE: I understand the design, but there is a mistake. If the new fact needs to be put at the very front of the list, then it is not possible to splice it in.

PROGRAMER: In that situation, BUCKET-INSERT will just CONS it onto the front of the list.

APPRENTICE: That will not work either. You want the fact to become a member of the bucket in the hash table, but this will not happen in the case where it is merely CONSed onto the front. The bucket in the table will still be the old list which is the CDR of the newly created list.

PROGRAMMER: OK, In that situation, BUCKET-INSERT will not only CONS the new fact on, but also store the result in the appropriate table entry.

APPRENTICE: That seems to fix the problem, however, there is a problem in INSERT. You said that the fact itself was to be put into each of the buckets hashed to. However, the way the code is written, what is inserted is FACT which is the atom being hashed on, and not the whole fact.

PROGRAMMER: You are right again. I should make the whole fact be an argument to INSERT1, and pass it to BUCKET-INSERT. Can you make that change?

APPRENTICE: Yes, here is what the code looks like now.

```
(DEFUN INSERT (FACT) (INSERT1 FACT 1 FACT))
(DEFUN INSERT1 (FACT POSITION WHOLE-FACT)
  (COND ((ATOM FACT) (BUCKET-INSERT WHOLE-FACT (HASH FACT POSITION TABLESIZE)))
        (T (INSERT1 (CAR FACT) (* POSITION 2) WHOLE-FACT)
           (INSERT1 (CDR FACT) (+ (* POSITION 2) 1) WHOLE-FACT))))
```

APPRENTICE: There is still a problem, from what I know about this kind of hash table, each fact should be in any one bucket at most once. However, I cannot prove that duplicates will not be inserted. In general, hashing causes collisions and neither INSERT1 nor BUCKET-INSERT does anything to prevent a fact from getting put in a bucket twice if it hashes to the same index twice.

PROGRAMMER: Ah yes. To fix this, I will break INSERT up into two parts: one, called INDEX, which determines the set (as opposed to the multi-set) of indices associated with a fact, and the other (still called INSERT) which will call BUCKET-INSERT in order to insert the fact into the corresponding buckets. [*And so the scenerio continues ...*]

This scenario illustrates several of the facilities the type of system we have in mind must provide. First, it must interact with the programmer during the design phase checking that the design is coherent and achieves its stated goals. Second it must record a representation of the logical structure underlying the design so that this may be used to detect bugs and guide evolutionary changes. Third, the apprentice must be able to recognize common design patterns within the code and to explain these in familiar, high level terms. It must also use these to structure its understanding of the design in ways which make it convenient to reason about the program and about proposed modifications to it.

III. Types of Programs -- Algorithms versus Systems

It is important to distinguish between two quite different kinds of programs: algorithms and systems. Each kind of program is important to software engineering. However, they present quite different demands and requirements. We argue that current program verification techniques are most useful and necessary for algorithms. The thrust of our work is directed towards the problems inherent in the design of systems.

Algorithms and systems differ along two primary dimensions: the character of their specifications, and the sources of their complexity. In general, an algorithm is a relatively short program which is precisely and concisely specified. For example, the Knuth-Morris-Pratt and the Boyer-Moore string matching algorithms each require roughly 100 lines of code but have a very short precise specification: the answer returned is the position of the first substring of the text which matches the input pattern. An algorithm is built to satisfy a precisely stated specification which has general utility. Therefore it is reasonable to expect that this specification will not have to evolve in the future. As a result, the effort required to actually verify the program can reap benefits far into the future. For example, Euclid's algorithm has survived unchanged for thousands of years.

In contrast to algorithms, software systems are large programs with specifications and other related documentation much larger than their code. More important, when specifying a system it is often impossible to state precisely what is to be done. Typically some claims are made about what must happen and others describe desirable but less crucial behavior. In any event these specifications often change, and the system is forced to evolve to meet the new criteria. The incompleteness and imprecision of the specifications for systems makes rigorous verification difficult, and the impermanence of the specifications reduces the rewards of producing such a verification.

The complexity of a typical algorithm stems primarily from clever underlying logic (often due to obscure optimizations) which *requires* proof in order to be believed. The intricacies of the string matching programs mentioned above would lead one to doubt whether they worked unless a rigorous proof were presented. If algorithms were subject to evolutionary change, this intricacy would be a significant liability.

In contrast, a system is usually made up of a large number of relatively small modules, each of which involves fairly routine code. An experienced programmer can easily understand and trust the local operation of such a system by recognizing standard patterns in the code. In other words, *recognition* can largely replace formal proof at this level. The complexity of software systems arises primarily from the number of interactions between modules. These are what make it difficult to assess the effect of a proposed change to the system. Systems tend to reach a point where the number of these interactions overwhelms unaided human abilities to manage them. From that point on, modifications become increasingly bug-prone.

These distinctions between algorithms and systems point to the need for different kinds of design aids in the two domains. The designer of algorithms needs proof checkers, theorem provers, and verification systems. While these serve a useful role for the system designer as well, they are not his bread and butter. Instead he needs tools which can help him evolve designs which satisfy evolving criteria. Rather than a tool for proving convoluted programs correct, a system designer needs a tool which can structure and remember the straightforward arguments for parts of large but routine programs so that the proofs can be used to guide an analysis of the effects of modifications.

IV. Problem Solving Theories

Three key ideas in current Artificial Intelligence theories of problem solving are: problem solving by recognition of the form of the answer, using planning in a simplified "abstraction" space in order to guide the problem solving process, and using debugging in order to transform an almost right solution into a correct solution.

One hallmark of an expert problem solver is the ability to recognize the *form* of the solution to a problem based only on a few high level features of the problem description. This reduces the initially unmanageable search in a very large solution space to an exploration of possibilities within a much smaller space. In electrical engineering, the form of a solution might be a particular circuit topology with certain components undetermined. In programming, the form of a solution might be a particular control strategy with unspecified primitive actions. This problem solving idea finds its antecedents in the Means-Ends analysis of [Newell, et. al., 1959] and in Minsky's notion of "islands" [Minsky, 1961] and was later formalized in the Planner programming language [Hewitt 1972] and its descendants Conniver [McDermott & Sussman 1974] and QA4 [Rulifson et. al. 1973] where the form of the solution is called a *plan.*

In sufficiently complex situations, a second paradigm called planning in an abstraction space, is also used. An abstraction space is a model of the real world in which some important details are intentionally omitted. Recognition of the form of the answer is first attempted in an abstraction space. If a plan is successfully formulated in the abstraction space, then it is modified to work in increasing more realistic spaces until a satisfactory solution is found. This problem solving paradigm was embodied in the ABSTRIPS program [Sacerdoti 1973].

Both the planning paradigm and the abstract modelling paradigm point to debugging as an unavoidable part of designing complex systems. The role of debugging in problem solving has been investigated by Sussman in his HACKER program [Sussman 1973]. When a plan is initially produced by recognizing the form of the answer in an abstraction space, the plan has associated with it an explanation of how it achieves its goals. However this "proof of correctness" is likely to be faulty because it depends on assumptions in the model which contradict facts in the real world. The almost-right plan is refined by developing a more realistic model of the situation and then using the old "proof of correctness" to guide the debugging process.

We believe these ideas constitute the best understanding to date of how people manage the complexity of planning and problem solving in complex domains and therefore these ideas should form the conceptual basis for developing computer aids for software engineering.

V. What Do Engineers Do?

In this section we discuss some specific techniques which have proved effective in more mature engineering domains such as electrical and mechanical engineering and which we think can be fruitfully applied to software engineering.

One might think that engineering is mainly concerned with the optimization of numerical parameters within physical systems and that computer science therefore has little to gain from the study of methodologies used in engineering. However, although engineers are at times concerned with numerical optimization, it is not their main activity. The dominant problem in engineering is the management of complexity during design and analysis. This can be seen in the following quote from a standard electrical engineering text [Bose & Stevens 65].

> A physical problem is never analyzed exactly. This is a consequence both of our inability to describe a physical situation completely and of the increasing complexity of the analysis as greater accuracy is demanded. A problem that involves events in the real world is always approached by making simplifying assumptions that hold only approximately, thereby forming a *model* of the events under study. The problem then reduces to that of analyzing the model. If the assumptions by means of which the physical situation was reduced to the model are reasonable, then our analysis should produce results that correspond to observed events, and the same type of analysis should be useful in predicting the behavior for other similar physical situations.

Thus, as the problem solving theories predict, engineers use abstract models to manage the complexity of their domains. Two particular abstraction techniques which engineers use are: the construction of multiple models each of which is accurate only under a restricted set of operating conditions, and the decomposition of complex systems into several possibly overlapping hierachical Organizations. Both of these techniques omit details which are not relevant to the task at hand. An example of the first technique is a linear model of a transistor which describes its behavior accurately only when it is operating within a certain range of frequencies and power. Sometimes several different models will be used which together form a good overall description, as for example the DC and frequency domain models for a circuit.

Engineers use decomposition to break up a large system into a (possibly overlapping) hierarchy of subsystems. Each subsystem is given a simple description which includes only those aspects of its behavior which are relevant to other subsystems. The whole artifact is then regarded as a loosely coupled network in which the behavior of the whole system may be deduced from the descriptions of the subsystems. The simplest kind of decomposition involves only a single non-overlapping hierarchy. However, sometimes a single component may be logically part of two or more different subsystems, and sometimes several different decompositions of a system are necessary in order to derive convenient descriptions for all of its behavior.

Decomposition is already a common technique in computer science. The use of subroutines as procedural abstractions described by their input-output behavior is well established. Data abstraction techniques allow a another kind of decomposition. Typically, these techniques are embodied in the features of a programming language such as CLU [Liskov et. al. 1977] or ALPHARD [Wulf 1974]. While we recognize the improvement such languages offer over earlier languages, we do not believe that they solve the whole problem. Convenient analysis frequently requires multiple decompositions of a single system, but unfortunately programming languages require that a system be represented by a single decomposition constained by the way in which the program is intended to execute.

The idea of problem solving by recognizing the form of the solution appears in engineering both in design and analysis. Evidence of this is the development of a vocabulary of useful macro structures which constitute the abstract forms of the solutions for broad classes of problems. In any engineering discipline, the basic units of design are a set of primitives (such as transistors, resistors, etc. or CONS, CAR, CDR, etc.) and rules for their legitimate combination. These generate an infinite number of legitimate combinations only some of which are useful. The macro structures in the intermediate vocabulary serve as stepping stones which make it cognitively feasible to derive the useful combinations from the primitives.

We do not intend to imply that there is a unique set of universally useful intermediate constructs but rather that it is always fruitful to look for them. Different domains employ quite different engineering vocabularies. Once an intermediate vocabulary is developed it expands the cognitive range of those practitioners who learn the vocabulary. As a result, they are capable of conceiving of yet more complex combinations, which leads to a higher level engineering vocabulary. For example, in electrical engineering one first learns to engineer useful networks using intermediate constructs such as voltage dividers. In order to combine these into more complex artifacts, one learns a higher level vocabulary including notions such as oscillators and amplifiers.

In programming the connection between the microscopic and the macroscopic is also mediated by an intermediate engineering vocabulary. If one is to work with a particular programming language one must know what its primitives do. However, program analysis which *exclusively* concentrates on the axiomatic description of program primitives is inadequate to deal with the complexity of real world programs. Indeed most of program understanding happens at a macro level which is more appropriate to the task at hand. It is at this level that one learns and remembers the useful patterns of doing things. For example, it is more fruitful to think about two linked lists, and about "splicing" as a kind of operation on these higher level objects, than to think of computer memory as a large collection of cells and about changing pointers in particular cells. In this way, we are much more likely to arrive at a computationally feasible and easily understandable description of the behavior of a program. One of our research goals is to create a catalog of intermediate engineering vocabulary for programming.

VI. Plans and Teleology

An engineer must have a representational system within which it is possible to utilize and coordinate information derived through the techniques described above. In most engineering disciplines there is a notion of the "design plan" which forms a skeleton around which all of this information is arranged. Of all the issues discussed so far, the design plan is the one least well addressed by other current work in computer science. Because the use of plans in software engineering is a central theme in our approach, we begin the presentation of our current work with an explanation of what a plan is and how it is represented.

In traditional engineering or software engineering, the behavior of a device or part of a device can be described in two ways. Some properties of a device are independent of its context of use. These properties constitute the *intrinsic* description of the device. For example, a capacitor can be described by the relation $I(t) = C\, dv(t)/dt$. The LISP function APPEND can be described intrinsically by its input-output behavior of returning the concatenation of its arguments. Intrinsic descriptions correspond to specifications in the literature of software engineering.

A device may also be described by its role in the plan for a larger mechanism. This is its *extrinsic* description. For example, a particular capacitor may be described as a coupling capacitor, a bypass capacitor, or a tuning capacitor, depending upon its purpose in the circuit. Similarly, APPEND may be used to produce the union of two disjoint sets represented as lists, or to attach a suffix to a root word represented as lists of characters. The abstract form of an answer retrieved in the process of engineering design is a plan in which each part is specified only by its extrinsic properties. Synthesis involves filling each role in the plan with a part whose intrinsic description satisfies the given extrinsic description.

A single part may have several extrinsic descriptions corresponding to multiple needs that it satisfies in the larger mechanism. For example, a screw in a camera may fasten two plates together and also provide a fulcrum about which to pivot a lever. There may also be several plans for a given device, describing its structure in different dimensions. In this situation, a part may fill several different roles in several different plans. For example, in a radio-frequency amplifier an inductor may be both part of a resonant circuit in the frequency domain plan and part of the bias network of a transistor in the DC plan.

The essence of understanding a mechanism is knowing the purposes of each part. This involves building a description of the mechanism which matches each part with its roles in the appropriate plans. Each role in each plan must be filled by some part of the mechanism and the intrinsic properties of that part must satisfy the extrinsic properties of its roles.

The utility of this kind of understanding is that it factors knowledge. A given plan fragment can appear as part of the plans for many different devices. Therefore understanding the logical structure of a plan fragment (which may be very difficult) need only happen once. Any properties of the plan fragment which can be proven, are known to hold wherever the plan is used. These plan fragments are the intermediate vocabulary items discussed in the last section.

VII. Representing Plans

In order to look at programs from the viewpoint of a design plan, we have devised a formalism called *plan diagrams* which can be used to describe both abstract program patterns and concrete programs. The basic entities in the plan diagram formalism are segments (input/output abstractions) and data objects. The formalism supports hierarchical description by allowing segments within segments (subsegments) and objects within objects (subobjects). The most basic relationship between these entities is the application of a segment to a set of input objects, yielding a set of output objects. The formalism includes four other primitive relationships: data flow, control flow, control splitting, and control joining. It is a straightforward matter to give the proof rules for the formalism, as has been done in [Rich & Shrobe 1976; Shrobe 1978].

In order to analyze programs written in a particular programming language one needs to have definitions for the language's primitives. We divide programming language primitives into two categories: connective tissue primitives such as IF-THEN-ELSE, WHILE, variables, argument passing, etc. which are concerned solely with implementing data and control flow, and actions such as arithmetic operations, CONS, CAR, CDR, etc. The first category is described by a translational sematics in which the primitive is mapped into the appropriate pattern of control flow and data flow links. Actions are represented as segments specified by pre-conditions and post-conditions. We have already constructed such language semantics for LISP [Rich & Shrobe 1976] and FORTRAN [Waters 1976,78] and have implemented systems which translate programs written in these languages into the plan diagram formalism. The translation process removes many of the surface features of the particular programming language, creating a flow graph which gives greater insight into the underlying logical structure.

Each segment in a plan is constrained either by its *spec-type* or by its *plan-type*. The spec-type of a segment is a formal statement of the relationships which are expected to hold for the input objects prior to its execution (pre-conditions) and the conditions which are guaranteed to hold immediately following execution of the segment (post-conditions). These conditions are expressed in a variant of the Situational Calculus of [McCarthy & Hayes 1969]. Each segment has associated with it an input situation and an ouput situation which are representations for the state of affairs on entry to and on exit from the segment.

The plan-type of a segment constrains what plan (i.e. what subsegments, and data and control flow) is used to implement the behavior specified for the segment by its spec-type. In the case of recursive programs and loops (which are represented as singly recursive programs) the plan-type for some subsegment will be the same as the plan-type for the overall segment.

Data objects are similarly described by object-type and plan-type. Object-types are a kind of data abstraction decomposing a data object into subobjects satisfying a specified set of contraints. Implementation rules constrain the plan-type of segments according to the plan-types of their input and output objects. The coordination of procedural and data abstraction is an important and novel feature of our represention system.

The plan diagram formalism is intended to facilitate our goal of cataloging the common and useful techniques of programming. The spec-types and object-types are arranged in a tangled hierarchy with more specific types inheriting descriptions from their super-types. For example, the LISP-style association-list is a specialization of both the object-type linked-list and the object-type associative-data-structure.

In order to represent the logical relationships in a plan, a plan diagram is augmented with a network of *purpose* links which summarize how the parts of a program interact in order to produce the behavior of the whole program. These links make it possible for a design aid system to explain how a program works, and reason about the potential effects of a modification. There are two basic ways in which the appropriate purpose links can be developed for a plan. They can be copied by reference to a stored plan in a catalog of programming knowlege (see section X), or they can be derived by reasoning directly about the plan itself.

Symbolic execution [Hewitt & Smith 1975; Hantler & King 1976; Rich & Shrobe 1976,78] of plan diagrams [Shrobe 1978] can be used to reason about programs and to create the appropriate purpose links. Symbolic execution of a plan operates as follows. A set of anonymous objects (skolem constants) is created, one object for each input to the outermost segment. Data objects are propagated along the data flow links leading to the initial subsegment. A subsegment is marked ready whenever all of its incoming data objects are present. The symbolic execution of the subsegment is then begun. This is done in one of two ways depending on whether it has a spec-type or plan-type.

If the subsegment is described by a plan-type its symbolic execution proceeds recursively. Its inputs are propagated along its data flow links to its subsegments and these are then executed as they become ready. If a subsegment is described only by a spec-type, it is first necessary to demonstrate that the subsegment's pre-conditions are satisfied. If this demonstration is successful, then the subsegment is applicable. Anonymous objects are created to represent the outputs of the subsegment and the post-conditions of the subsegment are asserted to hold in its output situation. The output objects are then propagated along data flow links to other subsegments which then become candidates for symbolic execution. Once all the subsegments have been executed, one then demonstrates that the assertions of the supersegment hold in its output situation. If this is successful, then the plan has been shown to achieve its desired effect.

The logical arguments which are constructed during this process are summarized into purpose links which capture the underlying teleological structure of the plan. There are two basic kinds of purpose links: *prerequisite* links, which show how the pre-conditions of a subsegment are satisfied by the interaction of the pre-conditions and post-conditions of other subsegments, and *achieve* links, which record how the pre-conditions and post-conditions of the various subsegments interact to achieve the post-conditions of their supersegment. These are similar to the proof summarizations used in [Moriconi 1977].

A plan may be thought of as an abstract program coupled with a logical analysis. However, it is important to note that this logical analysis need not necessarily be a "proof" in the sense of a guarantee of correctness. Our reasoning system [Shrobe 1978] is capable of conducting logical arguments which range from informal to rigorous. In many cases the plan for a program will only contain a "common sense" or engineering type analysis which is inadequate to guarantee correctness under all conditions, but which is good enough for purposes of explaining its teleological structure. When it is necessary, our reasoning system can be asked to carry out the verification of certain modules with full rigor. However, in this part of the process, we have made no advances over other verification systems. Our main goal is not the proof of correctness of large software systems, but rather an engineering oriented explanation and bookkeeping facility of some sophistication which will make it easier for a software engineer to modify a system while convincing himself that it does what he intends.

VIII. Temporal Abstraction

Temporal abstraction [Shrobe 1978; Waters 1978] is a modelling technique which makes it more convenient to analyze the logical structure of recursive plans. In a temporal model, the time behavior of a program is unfolded so that the occurrences of the subsegments can be regrouped to make common programming fragments more easily identifiable.

For example, consider the following recursive Lisp program which builds a list of the terminal nodes of a binary tree.

```
(DEFUN DEPTH-FIRST-FRINGE (TREE)
       (PROG (FRINGE) (DEPTH-FIRST-FRINGE1 TREE) (RETURN FRINGE)))
(DEFUN DEPTH-FIRST-FRINGE1 (NODE)
       (COND ((ATOM NODE) (SETQ FRINGE (CONS NODE FRINGE)))
             (T (DEPTH-FIRST-FRINGE1 (CDR NODE))
                (DEPTH-FIRST-FRINGE1 (CAR NODE)))))
```

We can analyze this program as the composition of three fragments:

(i) a tree-traversal segment, implemented by the depth-first plan, which enumerates the nodes of the tree,

```
(DEFUN DEPTH-FIRST-FRINGE1 (NODE)
       (COND ((ATOM NODE) ...
             (T (DEPTH-FIRST-FRINGE1 (CDR NODE))
                (DEPTH-FIRST-FRINGE1 (CAR NODE)))))
```

(ii) a filter segment which selects out the terminal nodes for further processing,

```
(COND ((ATOM NODE) ... NODE ...)
```

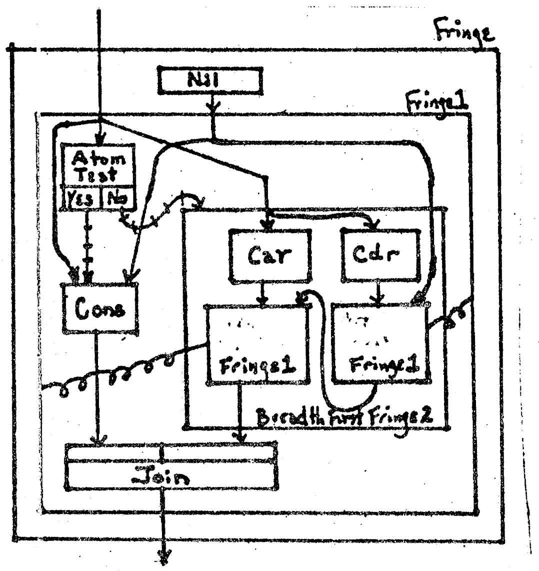
(iii) and an accumulation segment which builds a list of the selected nodes.

```
(PROG (FRINGE)  ...
       ... (SETQ FRINGE (CONS NODE FRINGE))
```

This intuitive decomposition into a generator, a filter and an accumulator has considerable conceptual power. This section will sketch the formalization of this decomposition used in the apprentice system. The method is be based on analyzing the history of *applications* during the course of an entire computation and grouping these into occurances of segments of like type.

Given a set of inputs to a plan diagram, the rules for symbolic evaluation unambiguously specify which of its sub-segments will be applied and to which arguments. Each such *application* is described by an input situation, a set of bindings of data objects with the input names of the sub-segment, an output situation, and a set of bindings for the outputs. The data and control flow links impose a natural partial order on the applications corresponding to their order of execution. A graphical representation of the plan diagram for Depth-First-Fringe is shown in Figure 1. (Cross-hatched lines represent flow of control, solid lines represent data flow. A box with two sections at its base is a test, one with two sections at its top is a Join. A curly line indicates that the inner segment is a recursive instance of the outer segment. Temporal abstraction makes it possible to model this program with the plan diagram of figure 2.

Temporal analysis begins with the notion of an *occurance set* of a particular plan (or spec) type. Given an application of a plan diagram the "occurance set of type1" consists of all applications within the plan diagram whose type is type1. In the Depth-First-Fringe program above there are three occurance sets of interest: (1) The occurance set of type Depth-First-Fringe1, (ii) The occurance set of type "Atom Test" and (iii) The occurance set of type "Cons". These correspond to the three fragments of code identified at the beginning of this section.



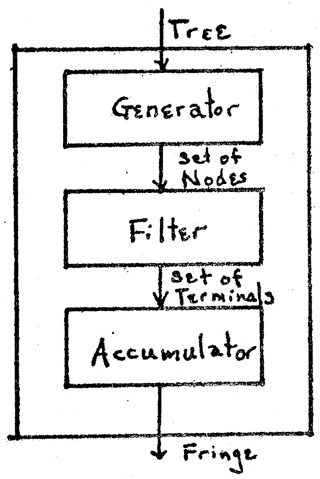


Figure 1: Plan for Depth-First-Fringe. Figure 2: Temporal model of Depth-First-Fringe.

Next we consider the sets of inputs and outputs of the segments within an occurance set. An occurance set consists of applications of segments of a common plan (or spec) type; the plan diagram for this type provides a set of local names for these segments' inputs and outputs. For example, the plan diagram for Depth-First-Fringe1 contains the input name "Node". It is useful to think of the set of objects which are bound to this name in any application of a segment of type Depth-First-Fringe1. Given an occurance set and a local name, we define a *temporal collection* to be a set of pairs consisting of (1) data objects which are bound to the selected local name and (2) the application in which they are bound. The temporal collection is partially ordered by the natural order of the applications. If Depth-First-Fringe is applied to the data object Tree-1 then the temporal collection of Node inputs to segments of type Depth-First-Fringe1 will consist of pairs containing all the nodes of Tree-1 in depth first order. This is shown diagramatically in Figure 3.

For every member of the occurance set of type Depth-First-Fringe1 there is a data flow link to an occurance of type Atom-Test. Such a collection of identical data flow links is called a *data flow bundle*. The collection of objects which flow along these links form two temporal collections, one at the segments on the initiating side of the data flow links and the second at the terminating side.

Using these concepts to examine the program Depth-First-Fringe makes it possible to decompose the program into units which can be analyzed by inspection. The occurance set of type Depth-First-Fringe1 is a *Binary Tree Traversal*; each segment in the set either has an atomic Node input, or it has data flow links to a CAR and a CDR segment which in turn have data flow links to other segments of type Depth-First-Fringe1. As already mentioned the temporal collection of Node inputs to members of this occurance set contains all the Nodes of the tree. There is a data flow bundle from this occurance set to the occurance set of type Atom Test. The latter set is a *filter*, an occurance set of identical test segments. Its input temporal collection contains all the nodes of the tree. Its output temporal collection is the sub-collection consisting of all Nodes which satisfy the Atom Test. These are the terminal nodes. There is a data flow bundle carrying this subset from the Atom-Test occurance set to the Cons occurance set. This last set is an *accumulation*; each segment in it takes one input from a previous Cons segment. The second input to each of these segments flows to it from a member of the Atom-Test occurance set. This decomposition is shown diagramatically in Figure 3.

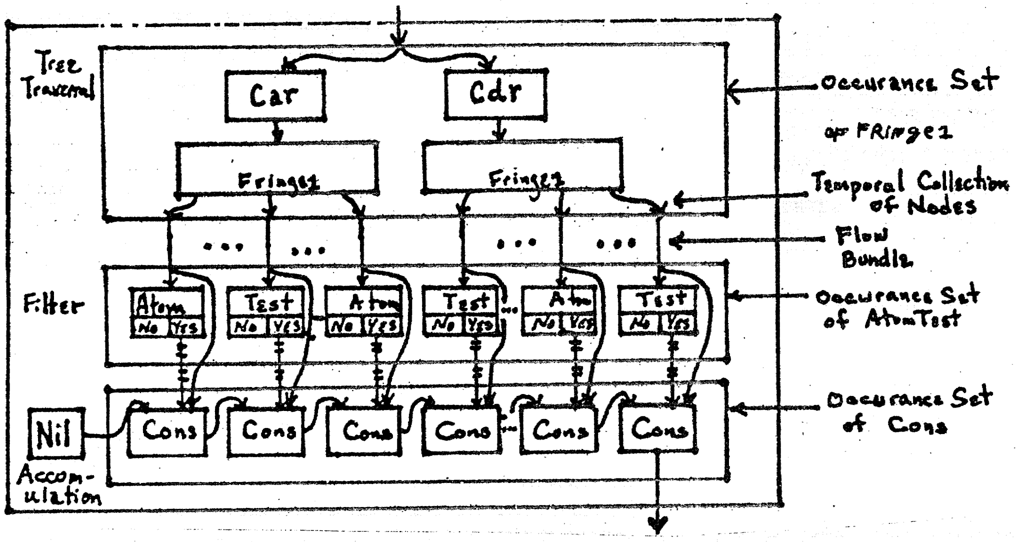


Figure 3: Temporal decomposition of Depth-First-Fringe.

We now build a model of Depth-First-Fringe. We model each occurance set as a single segment; each data flow bundle as a single data flow link and each temporal collection as a single data object. The resulting model consists of only three segments Binary Tree Traversal, Filter, and Accumulation. The data flow links in the model form a simple pattern, each segment taking a single input from its predecessor. From this viewpoint the program appears is seen as a simple composition. Notice that the temporal model suggests the following clear and concise explanation of Depth-First-Fringe: "The program consists of three steps, first it generates the nodes of the tree, second it selects out those nodes which are terminals, and third it builds a list of these nodes." This model is shown in figure 2.

IX. Plan Building Methods

The entire process of analyzing a program would be made much easier if it were possible to decide how to break a program into parts before determining what any of the parts do. One approach to this segmentation problem is taken by Waters [1978] who has developed and implemented a system which discovers the logical segmentation of a large body of common programs. His analysis is based solely on recognizing topological patterns of data flow and control flow without regard for the specifications of the various operations involved. These patterns are called *plan building methods* (PBMs), because they can be thought of as instructions for how to combine plans together to form more complex plans.

The simplest PBMs correspond to the standard structured programming notions of conjunction, composition, and conditional. More complex PBMs decompose recursive plans by making use of temporal abstraction and trajectories.

The recursive PBMs make it possible to construct a temporal model for a recursive program in which its structure is revealed as the composition of standard segments which can be understood in isolation from each other. Three basic recursive PBMs produce three types of standard recursive segments: terminations, filters, and augmentations.

A filter segment is one which tests a temporal sequence of values and selects out a consistently ordered subset to be acted on by other segments.

A termination segment is one which tests some temporal sequence of values of values and can cause the termination of the recursive program as a whole. (The ATOM test in the DEPTH-FIRST-FRINGE program doubled as both a filter and a termination segment). As such, it determines the length of all the trajectories in the temporal decomposition of the given recursive plan, but does not affect what is computed in these trajectories.

All other segments are referred to as augmentations. Augmentations take in trajectories of values and perform calculations in order to create additional trajectories of values. Typically, an augmentation will have feedback of data flow to itself, so that it can utilize past values in its computations. The accumulation segments in Section VIII are augmentations.

The analysis of a recursive plan using PBMs turns out to be straightforward. The basic idea is that a part of the program affects the rest of the program only if it either has data flow to some other part of the program, or controls when some other part of the program will be executed. Decomposition is achieved by locating segments of the plan which do not affect anything in the rest of the plan. (Note that this has to be weakened slightly in the case of terminations.) When such a segment is found, it is pulled out of the plan and the process is repeated until nothing else can be pulled out. The segments that are pulled out are connected together temporally by trajectories as explained in Section VIII. Thus, Waters' analysis provides one means of decomposing systems into loosely coupled sub-systems.

An experiment was performed in order to determine whether or not the particular PBMs chosen had a wide range of applicability. A random sample of 20% of the programs in the IBM Scientific Subroutine Package were analyzed in terms of PBMs by hand. All of the programs turned out to be analyzable in terms of the the PBMS. More importantly, nearly 90% of the time, the analysis broke the programs up into segments which were so simple that it was trivial to understand what the segments themselves were doing.

X. A Library of Plans and Analysis by Inspection

As we have seen, one of the major goals of temporal abstraction and PBM decomposition is to facilitate analysis by inspection. The basic idea is to analyze a given program by recognizing patterns of segments in the decomposed plan as instances of commonly known correct plans stored in a library. Work such as [Barstow 1977] suggests that it is possible to catalog substantial portions of programming knowledge in a reasonably concise formalism. We have begun a similar cataloging effort [Rich forthcoming] using the plan formalism, which we believe will have several advantages. Most important among these is the fact that our plan library is not biased towards either synthesis or analysis, but attempts to capture the knowledge underlying both.

The plan library is a formalization of the intermediate vocabulary (Section IV) of software engineering. It includes standard plans (patterns of data flow and control flow between specified subsegments) for implementing common input-output specifications, and standard plans (sets of objects with constraints between them) for implementing common data abstractions. Plans in the library are pre-proven, i.e. they have attached to them explanations that can be combined with the explanations of other plans in order to arrive at a complete explanation of how a given program works.

Examples of plans that we have formalized are: (data plans) implementing a set as a list, implementing a binary relation as a hash table, implementing a stack as a sequence plus a cell, implementing a tree using pairs; (procedural plans) list traversal, tree traversal, filtering, linear search, sequential accumulation. Many of these plans fit into a specialization hierarchy which aids in finding the right plan during analysis or synthesis. For example, binary tree traversal is a specialization of tree traversal, and hash tables are a specialization of associative data structure. One of our research goals is to extend this catalog to include even higher level concepts, such as interpreter, data-based system, etc.

Recognition of the PBMs was so successful because there were a small number of them and they were all very different from each other. Recognizing instances of library plans in the plan for a given program will be more difficult because there are many plans in the library and they tend to have some features in common. Our first approach will be to see how far we can get continuing the bottom-up style of recognition from PBM analysis into recognition of the intermediate vocabulary. However, we will certainly have to eventually develop some top-down recognition strategies to make it possible to recognize the very high level concepts.

XI. Dependency Directed Reasoning and Program Modification

A prototype dependency directed reasoning system [Shrobe 1978] has been implemented in the AMORD programming language [de Kleer et. al. 1977]. In a dependency directed system every new assertion entered into the data base is accompanied by a *justification* stating which other assertions form the logical support for the new one. The justification itself is an object which the system can inspect and manipulate.

Assertions in the reasoning system have two states: *in* or *out*. An *in* assertion is one which is believed. An *out* assertion is one not currently believed. A special module called the Truth Maintainence System (TMS) [Doyle 1978] is responsible for guaranteeing that all assertions with valid reasons to be believed are *in* and all assertions which lack valid justifications are *out*. This facility is particularly flexible because an assertion can be justified by the lack of valid support for some other assertion. Technically this means that the assertion F1 may have a justification which depends on the *out*ness of some other assertion F2. This amounts to saying that as long as there is no reason to believe F2 one should assume F1. If reason to believe F2 is ever discovered, the TMS will automatically bring F2 *in* and F1 *out*. Addition of an assertion (F2) can cause another assertion (F1) to become invalid. Logics with this property are called *non-monotonic* [Doyle 1978]. The semantics of such logics is discussed in [McDermott & Doyle 1978].

We see two key applications for dependency directed reasoning in software engineering: hypothetical reasoning during theorem proving and analysis of program modifications. For example, [Shrobe 1978] describes the use of dependency directed reasoning to reason about side-effects by first assuming that the degree of sharing between complex data structures is limited. Various desired properties of the program are then proven under this simplifying assumption. Sometimes such a cursory analysis is all that is appropriate. However, when a more careful exploration is desired, the assumption can be removed and replaced by a more cautious assumption or by no assumption at all. In many cases, some of the important properties of the program do not depend on the assumption and remain *in*. However, if some property does in fact depend on the assumption it will go *out* indicating that the original proof is no longer valid under the conditions of sharing. A more complicated proof of that property can then be attempted.

A dependency based reasoning system also makes it possible for incremental changes in a program to necessitate only incremental changes in the analysis of the program. Suppose, for example, that a programmer decides to change the representation of some data object from arrays to binary trees. He would then replace all instances of loops enumerating the elements of the array with tree traversals enumerating the nodes of the tree. Although the new code might bear little superficial resemblance to the old code, [Shrobe 1978] shows that dependencies make it possible to handle this change by an incremental re-analysis of the program rather than by a new analysis from scratch.

[Doyle 1978] shows how dependencies can be used to achieve many of the control disciplines which have been used in automated theorem proving systems. Most important among these is *dependency directed backtracking* in which a contradiction is removed from the system by first identifying those assumptions which lead through a chain of dependencies to the contradiction. A new set of dependencies is constructed which guarantees that all of these assumptions cannot be *in* at the same time. The reasoning system can then select an assumption to reject. [Stallman & Sussman 1977] show how this technique reduces combinatorial explosions in an electrical circuit analysis system. [Shrobe 1979] shows that dependencies are adequate to achieve the effect of the contexts of QA4 [Rulifson et. al. 1973] and Conniver [McDermott & Sussman 1974] while avoiding some of their problems.

XII. Conclusion

We intend to furhter develop the modules we now have in order to implement and experiment with a computer system that can understand and reason about programs using the methods and representations presented above. This system should be the prototype for an interactive programming environment in which both the computer and the human programmer cooperate to produce software more quickly and reliably than either could do working alone. In this environment the programmer will treat the computer as if it were a colleague, explaining and developing the program design interactively. The computer will play a passive role; its strength is not design but rather careful bookkeeping and criticism.

Ours is only one of many approaches towards alleviating the currrent software crisis. High level languages, structured programming, verification, and automatic programming also make claims to being part of the solution. Indeed, a pluralism of approaches seems both warranted and necessary. However, with the exception of [Moriconi 1977], our work appears to be the only project which directly confronts the issue of incremental and evolutionary design of large software systems.

In contrast to automatic programming and the most ambitious high level languages, the design aid approach allows an important simplification, namely that the computer need not be an expert in questions of efficiency. If languages, compilers, and automatic programming systems are to raise the level of abstraction of programming significantly, thereby hiding efficiency considerations, then they themselves must possess an expertise in efficient implementation resonably close to that of a competent programmer. Otherwise, they will not be used. A design aid system, in contrast, can provide significant assistance with virtually no understanding of efficiency at all. Furthermore, as techniques for reasoning about efficiency are developed they can be added to the system. There is always the escape hatch that the programmer can modify automatically generated code without losing the benefits of the design aid. Thus our approach allows a smooth transition from a passive assistant to a more automatic system.

We also set a more modest research goal in comparison to program verification. We do not seek to guarantee the correctness of a large software system, a task we feel is very much harder than what current technology can manage. We set instead an intermediate goal of *understanding the structure* of the system and using this as a guide to bookkeeping and checking during design and modification. We expect that as our technology for automatically deducing the logical structure of programs progresses it will find application in the area of automatic verification. This should eventually make it possible to construct verifications (perhaps even for large systems) which are more intelligibly structured than presently is the case.

Bibliography

Barstow, David [1977], "Automatic Construction of Algorithms and Data Structures", PhD. Thesis, Stanford University, September 1977.

Bose, Amar G. and Stevens, Kenneth N. [1965], "Introductory Network Theory", Harper & Row N.Y., 1965, page 1.

de Kleer, J.; Doyle, J.; Steele, G.; and Sussman, G.J. [1977], "AMORD: Explicit Control of Reasoning", Proc. of the Symp. on AI and Programming Languages, August 1977.

Doyle, Jon [1978], "Truth Maintenance Systems for Problem Solving", MIT/AI/TR-419, January 1978.

Hantler, Sidney and King, James C. [1976], "An Introduction to Proving the Correctness of Programs". Computing Surveys V8 #3, September 1976, pp. 331-353.

Hewitt, Carl [1972], "Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language For Proving Theorems and Manipulating Models in a Robot", MIT/AI/TR-258, April 1972.

Hewitt, Carl and Smith B.C. [1975], "Towards A Programming Apprentice", IEEE Trans. on Soft. Eng., V1 #1, March 1975.

Liskov, B.; Snyder, Alan; Atkinson, Russell; and Schaffert, Craig; [1977], "Abstraction Mechanisms in CLU", CACM, August 1977, pp. 564-576.

McCarthy, J. and Hayes, P. [1969], "Some Philosophical Problems from the Standpoint of Artificial Intelligence", Machine Intelligence 4, American Elsevier, NY, 1969.

McDermot, Drew V., and Sussman, Gerald J. [1974], "The CONNIVER Reference Manual", MIT/AIM-259a, January 1974.

Mcdermott, Drew V. and Doyle, Jon 1978, "Non-Monotonic Logic I", MIT/AIM-468, August 1978.

Minsky, Marvin [1961] "Steps Towards Artificial Intelligence", Proc. IRE, Vol. 49, No. 1, 1961.

Moriconi, Mark [1977], "A System for Incrementally Designing and Verifying Programs, ISI/RR-77-65, November 1977; also PhD Thesis Univ. of Texas, 1977.

Newell, A. Shaw, J. C. and Simon H. [1959] "Report on a General Problem Solving Program", Proceedings of the International Conference on Information Processing, Pairs UNESCO House 1959.

Rich, C. [forthcomming], "A Library of Programming Plans with Applications to Automated Analysis, Synthesis and Verification of Programs", forthcoming PhD thesis, MIT Cambridge MA, expected 1979.

Rich C. and Shrobe H. [1976], "An Initial Report on a LISP Programmer's Apprentice", MIT/AI/TR-354, December 1976.

Rich, C. and Shrobe, H. [1978], "Initial Report on a LISP Programmer's Apprentice", IEEE Trans. on Soft. Eng., V4 #6, November 1978, pp. 456-467.

Rulifson, Johns F.; Derksen, Jan A.; and Waldinger, Richard J. [1973], "QA4: A Procedural Calculus for Intuitive Reasoning", SRI AI Center Technical Note 73, November 1973.

Sacerdoti, Earl D. [1973], "Planning in a Hierarchy of Abstract Spaces", IJCAI-73, August 1973, pp. 412-422.

Shrobe, Howard E. [1978], "Reasoning and Logic for Complex Program Understanding", PhD thesis, MIT, August 1978.

Shrobe, Howard E. [1979], "Explicit Control of Reasoning in the Programmer's Apprentice", To be presented at the 4th Workshop on Automated Deduction, February 1979.

Stallman, Richard and Sussman, G.J. [1977], "Forward Reasoning and Dependency-Directed Backtracking In A System for Computer-Aided Circuit Analysis", Artificial Intelligence Journal, October 1977.

Sussman, G.J. [1973], "A Computational Model of Skill Acquisition", MIT/AI/TR-297, August 1973; also appeared as "A Computer Model of Skill Acquisition", New York, American Elsiver 1975.

Waters, Richard C. [1976], "A System for Understanding Mathematical FORTRAN Programs", MIT/AIM-368, August 1976.

Waters, Richard C. [1978], "A Method for Automatically Analyzing the Logical Structure of Programs", PhD thesis, MIT Cambridge MA, August 1978, (to appear as MIT/AI/TR-492).

Waters, Richard C. [1979], "A Method for Analyzing Loop Programs", to appear in IEEE Trans. on Soft. Eng., in 1979.

Wulf, W.A. [1974], "ALPHARD: Towards a Language to Support Structured Programming", Carnegie Mellon Univ., April 1974.