MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Working Paper 158 January 1978

PLAN VERIFICATION IN A PROGRAMMER'S APPRENTICE

(Ph.D. Thesis Proposal)

by
Howard Elliot Shrobe

Brief Statement of the Problem:

An interactive programming environment called the Programmer's Apprentice is described. Intended for use by the expert programmer in the process of program design and maintenance, the apprentice will be capable of understanding, explaining and reasoning about the behavior of real-world LISP programs with side effects on complex data-structures. We view programs as engineered devices whose analysis must be carried out at many level of abstraction. This leads to a set of logical dependencies between modules which explains how and why modules interact to achieve an overall intention. Such a network of dependencies is a teleological structure which we call a plan; the process of elucidating such a plan stucture and showing that it is coherent and that it achieves its overall intended behavior we call plan verification.

This approach to program verification is sharply contrasted with the traditional Floyd-Hoare systems which overly restrict themselves to surface features of the programming language. More similar in philosophy is the evolving methodology of languages like CLU or ALPHARD which stress conceptual layering.

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

Working Papers are informal papers intended for internal use.

© MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1978

Abstract

In this proposal we describe an interactive programming environment to be used as a tool by the expert programmer in the process of program design and maintenance. This environment (called the Programmer's Apprentice) will be capable of understanding, explaining and reasoning about the behavior of real-world programs, with particular emphasis on LISP programs involving side effects on complex data-structures. In achieving such a system, our pivotal theoretical committment will be to a view of programs as engineered devices whose analysis can be carried out at many level of abstraction. The analysis of a program will lead to a set of logical dependencies between modules which explains how and why modules interact to achieve an overall intention. Such a network of dependencies is a teleological structure which we call a plan; the process of elucidating such a plan stucture and showing that it is coherent and that it achieves its overall intended behavior we call plan verification.

Our research will be concerned with the design of a plan verification program called REASON. Given a description of the data flow between modules (specified at any level of abstraction), REASON will determine whether or not these modules cooperate to achieve their intended net behavior. In so doing, REASON elucidates the plan structure which, in turn, is used as the pivotal data structure in the pertubation analysis necessary for program evolution, explanation and debugging.

This approach to program verification will be sharply contrasted with the traditional Floyd-Hoare systems which overly restrict themselves to surface features of the programming language. More similar in philosophy is the evolving methodology of languages like CLU or ALPHARD which stress conceptual layering in program structure and therefore have a verification methodology more like our own.

Finally, a methodology of program design will be explored in which man and machine interact in the formation of a verified plan at an appropriate level of abstraction. It is hoped that examination of this process will shed light on the theory of automatic design, thereby aiding in the future development of rich automatic programming systems.

The Problem of Program maintenance - Notes of A Beleagered Systems Programmer

It is now a commonplace that software, in particular software maintenance, is the major expense of computation. As machines have grown larger and faster the programs which run on those machines have grown more ambitious and complex. Unfortunately, the tools for maintaining software have not kept pace with this growth.

This leaves the programmer or designer of large systems in a bind. Specifications for large systems are frequently incomplete; simply put, the user doesn't know exactly what he wants. Given fuzzy criteria the designer does the best he can, guessing here, making temporary choices there. In addition the art of programming is at best just an art, a well developed engineering discipline does not yet exist.

Once a program reaches the stage of initial implementation, however, new desiderata are almost always discovered. "This report should have these 3 extra fields, that one provides extraneous information." New hardware becomes available resulting in changes in the requirements and new opportunities for improvements. In addition, the currently available features suggest new ones which could be implemented if only certain modifications were made.

So while the first implementation is running, work is started on adding features and reworking the last implementation. Running experience reveals the existence of some new bugs which force additional redesign. In this process the programmer again and again finds himself trying to remember whether it is safe to smash the record before it is stored, whether any module is using the second bit of the dispatch queue entry, etc. In general he is forced to consider all possible places which might be effected by any proposed change. Of course, one does what one can and version two eventually appears.

At this point, the user and the programmer notice that there are five new features, a brand new terminal which would allow real time interaction, and of course the inevitable bugs. So while version two is being run, version three is being laid out on the drawing board. And so on...

Good programmers use several devices to try to keep such problems under control. First, as much as possible they try to demonstrate to themselves that the programs they have written do work as desired. Even when the specifications are incomplete, the programmer will attempt to do this to the degree allowed by the specifications he does have.

A second technique used by expert programmers is careful documentation and

commenting of program code. By writing out explicitly all design decisions and implementation strategies and by noting in the code itself how various statements depend on these basic parameters, the programmer hopes to gain some control over the complexity of the process.

There are, however, two limitations which seriously impair the usefulness of this process. First, a complete listing of all such dependencies would be orders of magnitude larger than the code, a prospect not pleasant to consider. Secondly, even with all such commentary the problem would be in finding the right comments at the right times. The printed program listing is not appropriately indexed for the kinds of tasks we are describing. Nor for that matter does human memory seem to be well suited to this task (as anyone who forgot that last change of cadr to caddr can well attest).

The programmer's apprentice approach to this problem is an attempt to create an interaction between programmer and programming environment in which the above problems become more tractable. In essence, our view is that if the programmer could communicate to this environment the information normally given as documentation and commentary, then it would be possible for the programming apprentice to fill in the rest of the details, indexing this information so as to be useful in explanation and evolution. In addition, it seems that it should be possible to build a library of standard (or even idiosyncratic) programming concepts such that the interactions could be brief enough to be usable in a real-world situation.

Thus a typical interaction between programmer and apprentice might be something as follows (Note: as usual the use of English is a convenient fiction):

Programmer: The symbol table for this section will be done using a LISP style hash table, indexed by symbol. The cadr will hold the relevant block-ids for this symbol.

Aprentice: OK

Programmer: To mark a symbol present in a block, I'll do the following. If the symbol is already defined then I'll fetch it from the table and add the appropriate block ID by side effect, otherwise I'll build the new record and insert it.

Apprentice: What does it mean for a symbol to be defined?

Programmer: Present in the hash table.

Apprentice: OK

Programmer: Here's the code for marking a symbol present in a block.

```
(defun mark-present (symbol block-id commentary)
       (prog (entry)
             (cond
              ((setq entry (lookup symbol))
               (rplaca (cdr entry)(sorted-insert block-id (cadr entry))))
              (t (setq entry
                       (insert (list symbol (list block-id)) commentary))
             (return (cadr entry))))
```

Apprentice: What is commentary?

Programmer: It's the third field of a symbol table record and it's idiosyncratic.

3 Months later

Programmer: I'm going to change hash-table entries so that there is no commentary field and so that the car is the symbol and the cdr is the list of block-ids.

Apprentice: OK, you'll obviously have to change the entry-building code in hash-table insert but also you'll have to make a change in mark-present where you add a new block-id by side-effecting in a new list of block-ids. In the old version this worked since the block-id list was the car of the cdr. Now it's the cdr itself and doing the rplaca will make the new block-id list be a member of the old block-id list rather than accomplishing the correct replacement.

Programmer: Right, change it to:

```
(rplacd entry (sorted-insert block-id (cdr entry)))
```

Apprentice: OK. Also the compiler routine Present-In-Block needs fixing.

etc.

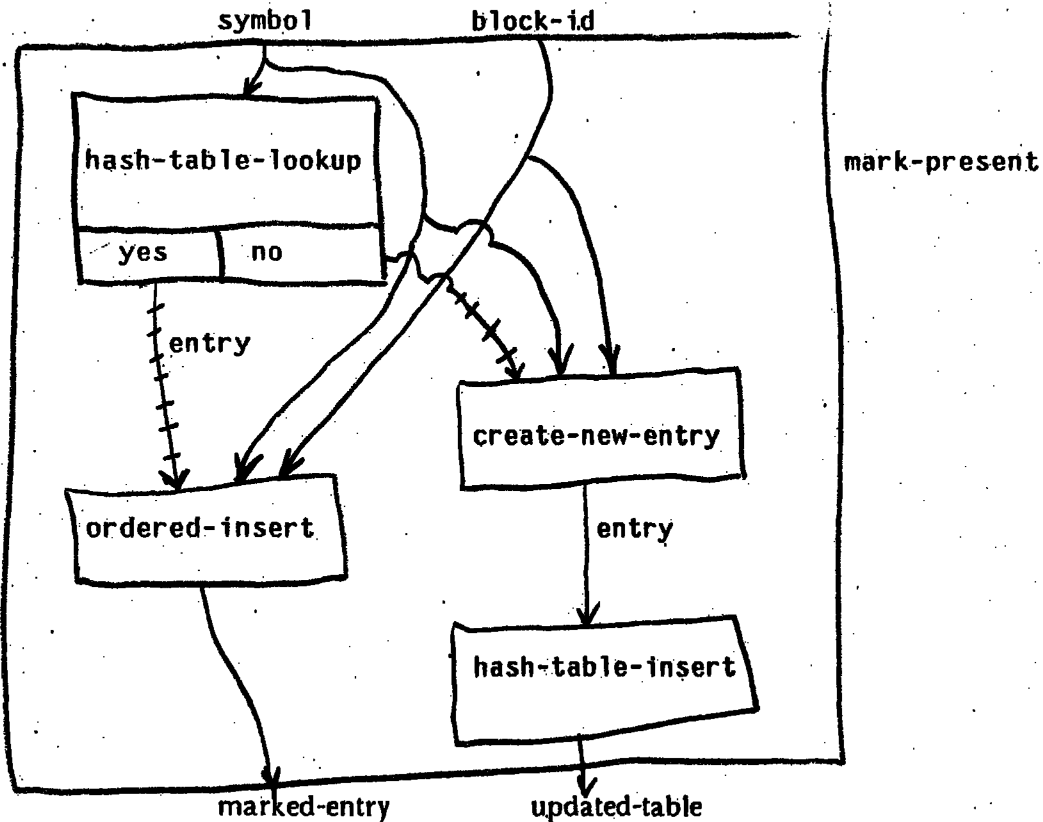
Programs As Engineered Devices

In writing programs, as in building bridges or constructing electronic circuits, the engineer will divide his problem into subunits and attempt to solve each as independently as possible. A radio for example will be divided into r-f and a-f stages each of which has its own internal structure. The communication between these two sections is expected to be kept to a minimum.

Similarly a programmer creating an air-line reservation system will divide his problem into terminal handling, data-base maintenance, etc. Each of these problems is itself quite complex and has internal structure in its solution. The unifying element between these separate sections of the program is in the communication discipline between them, in the assumptions about overall structure, and in the common use of sub-modules to solve different internal problems.

To be more specific, consider the program referred to above. The overall problem is apparently a compiler for a block-structured language. A hash table is used in implementing the symbol table of the compiler. To achieve a simple action such as marking a symbol with a block-id several other operations such as hash-table-lookup, sorted-insert, rplaca, etc. are called upon. These sub-actions interact in a purposeful manner to achieve the desired goal, namely that the symbol table indicate that the specified symbol is defined in the indicated block.

We might diagram this as follows:



Thus, the mark-present operation is accomplished by a pattern of interactions among other operations. In the process of program development such accumulation of behavior is done both bottom up and top down, i.e. sometimes one imagines the existence of a module which will do a needed job even when that module does not yet exist; at other times already existing modules will suggest new behavior which seems useful. While we are not proposing a methodology of program construction, we are noting that to be comprehensible at all a program must have some sort of modular structure.

Notice that many of the modules used to build mark-present have internal structure of their own. Ordered-insert, for example, will probably consist of a search-loop, a cons, and a rplacd. The hash-table routines will involve steps such a hash, bucket-fetch, etc. Thus, the structure given above is a layered one. There will be boxes within boxes until one finally reaches programming language primitives.

In any engineered device there will be communication between the modules. In mechanical devices this is done by physical contact, in electronics by wiring. In programs, communication between modules is accomplished by the control and data-flow primitives of the programming language. But in any engineered device there are limiting requirements on the communication between modules. Components of a circuit will have voltage or current limits which they expect to be respected by the component at the other end of the wire; similarly mechanical devices expect forces to be within specified ranges.

Programs expect their communicating partners to respect certain limits as well, although the types of pre-conditions are quite general. A segment (our name for a program module) may require that its input objects be well formed objects of a specific type, or it might require something more elaborate of them like being sorted by primary and secondary keys. Such pre-conditions are termed expectations.

A module can be thought of as promising that certain conditions will hold after its execution as long as its expectations are satisfied. Thus, an amplifier will promise to deliver a stronger signal and a sort routine a sorted list as long as the inputs meet the appropriate requirements. We call such guarantees on output conditions assertions. Thus, a module may be abstractly described by its input-output behavior; i.e. by pairs of expectations and output assertions.

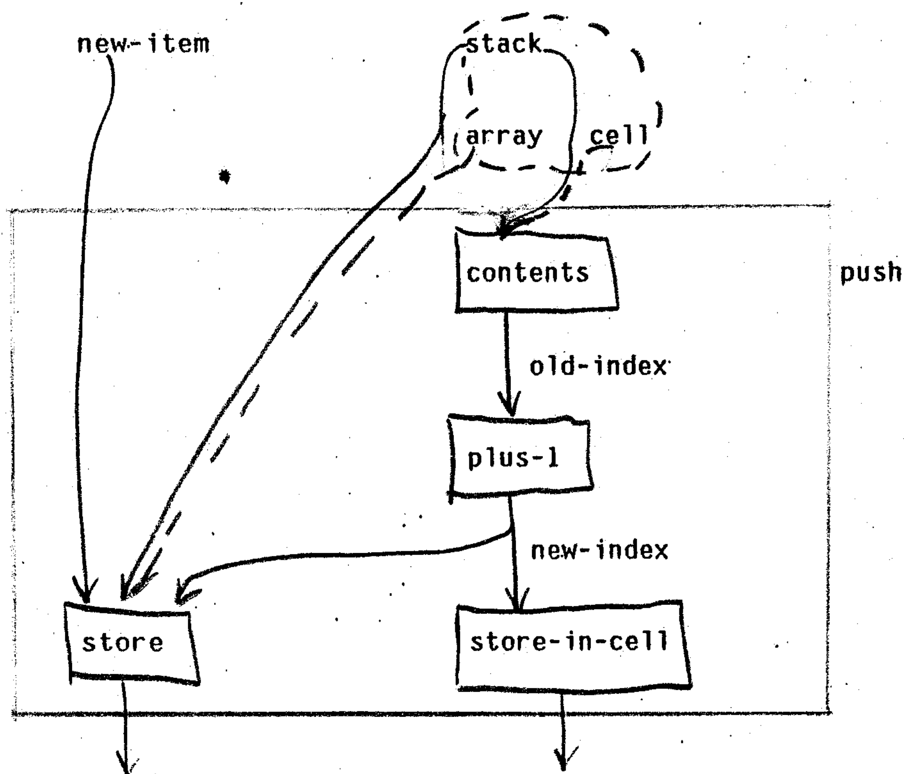
For an engineered device to function at all, it is necessary that the pattern of interactions between sub-modules be such that every module's expectations be satisfied at the time of its invocation. Further, the pattern of interactions must be such that the desired overall result is a logical consequence of the accomplishments of all the sub-modules. Such logical links between sub-segment behavior we term purpose links; those links which explain how a sub-module's expectations are met are called pre-requisite links, those which explain how the overall intentions are met are called achieve links. The pattern of purpose links is what we term a plan and the process of deriving such logical links from the connective pattern alone is called plan verification (in the case of programs this connective pattern is given by the data and control flow).

As stated above, the logical connections between modules involve not only the abstract description of modules but also the overall commitment to design strategy. For example, consider the symbol table routine above. To conclude that the symbol is properly marked with the block-id one must know that a design choice has been made to do this using hash-tables as opposed to property lists or some other scheme. Thus, purpose links can be seen to be logical patterns involving abstract descriptions of both the sub-modules and of

the operant design choices.

Design choices are schemes for building abstract behavior and objects using other less abstract objects as the building blocks. This may be seen by considering a scheme for implementing a stack using an array and a cell.

Conceptually, a stack is an object with a TOP-ELEMENT and a HISTORY. It has PUSH and POP operations. One implementation scheme uses an array and a cell. The top element of the stack is the array item indexed by the contents of the cell. The history part of the stack is the sub-array from 0 up to the item indexed by the contents of the cell. Push is accomplished by adding one to the contents of the cell, storing this number in the cell; and the new top item in the array item indexed by this number. This may be diagrammed as follows:



We would want to show that the new top is new-item and that the new history consists of all the old items plus the old top (in appropriate order). This will clearly require making reference to the implementation pattern described above, that is, showing that the new top is new-item will depend on the description of plus-1, contents, and the fact that (within this design scheme) the stack-top is the item indexed by the contents of the cell.

Similar conceptions occur in electronic devices where one may chose to regard a signal as either a current or a voltage. The internal structure and logic of the amplifier for such signals varies according to this choice, although the conceptual behavior of amplifying a signal remains true in either case. It is one of the key problems of design that any such implementation strategy imposes restrictions which were not implicit in the conceptual object being implemented. Using arrays imposes finite size on the stack; different amplifier designs impose different restrictions on power and signal range.

In summary, the plan of an engineered device is a set of logical connections between the conceptual descriptions of sub-modules, the descriptions of implementation strategy, and the overall intentions for the device being engineered. These logical steps explain how each module of the overall device contributes to the higher level conceptualization as well as why each sub-module is capable of functioning. The lack of such a logical connection in a proposed device would indicate a conceptual failure or design bug.

Given that modules of a device may themselves be conceptual constructs with internal structure, plans provide an abstracting mechanism describing the structure of the device at a level appropriate to the task at hand. Plans also allow one to describe and reason about the behavior of incompletely designed devices, since a module's net behavioral specifications may be used within a larger plan even if there is as yet no internal plan to accomplish the behavior of the sub-module.

Although we believe the research we will conduct will have bearing on any engineered device, it is necessary to focus attention in order to make progress. For the remainder of this document we will consider plan verification only within the context of programming; others are conducting similar research in other disciplines <Stallman & Sussman 1976>; a rich cross fertilization of ideas and techniques has and doubtlessly will continue between these areas of exploration.

The Use of Plans in Program maintenance and Explanation

Plans, as outlined above, give a teleological description of program behavior, abstracted to a level of description which is convenient to the programmer. It is, thus, a rather trivial matter to generate explanations of a program from a plan. Since plans contain more information than does the program itself, such explanations will be richer than a mere recitation of the code. This has been explained in Rich & Shrobe <Rich & Shrobe 1976>.

Of more interest to future research efforts is the notion of using plans as a means of program modification or evolution, i.e. in helping the beleagured systems programmer referred to above. As we have noted, plans capture the relationship between program design choices, abstract modularization, and overall intentions. In doing this, they localize the effects of changes in design strategy, and specify the teleological requirements which must be satisfied in any modification of the design.

As a simple case consider a hash-table insert routine which has been implemented using ordered linked-list buckets with a count field. The code for such a program might be:

```
(defun insert (item)
       (insert-in-bucket (table (hash (key-part item))) item))

(defun insert-in-bucket (bucket item)
       (do ((previous-list bucket (cdr previous-list))
            (current-list (cdr bucket) (cdr current-list)))
           ((null current-list)(rplacd previous-list (list item)))
           (and (greater-than (car current-list) item)
                (rplacd previous-list (cons item current-list))))
       (rplaca bucket (1+ (car bucket))))
```

Suppose that for space efficiency, it was desired to change to a rehashing scheme. Since this change is strictly a design issue dealing with buckets, the plan would tell us that the structure of the insert module itself is correct, but that the insert-in-bucket module as well as the communication between the two modules might require change. It could further tell us that the last line (i.e. the rplaca which bumps the count) is no longer relevant. A simple system might stop at this point.

However, a full blown system would be expected to realize that there is still common structure between the old and the new design. For example, buckets in both approaches are LINEAR-OBJECTS searchable by a LINEAR-SEARCH-LOOP. The difference is in the nature of the BUMP, EXHAUSTION, and TERMINATION steps. In the rehash scheme, BUMP is of course the rehash operator and termination of the search is indicated by a special marker (such as nil) indicating that a slot is free. Exhaustion of the search might be indicated by the rehash routine returning a negative number. Also making an item a member of a bucket in the rehash scheme is simply a matter of inserting in the array. Thus, an advanced system might guide the programmer to the following new design.

```
(defun insert (item)
       (insert-in-bucket (hash (key-part item)) item))

(defun insert-in-bucket (initial-slot item)
       (do ((slot initial-slot (rehash slot)))
           ((minusp slot)(error 'no-slots-left))
           (and
            (null (table slot))(store (table slot) item))))
```

Notice that this sort of full-blown petubation analysis included not only the plan itself but also the structure of knowledge about program design. The fact that both the old and the new style buckets were linear-objects indicated the similarities between the two search routines. Such structure is captured in a knowledge base which uses plans as part of its representation. However, the plans in the knowledge base are typically quite abstract (such as the plan for search loop) and so they serve as prototypes for program structure.

Our Proposed Plan Verification System

This section will describe a plan verification system called REASON which adequately addresses the problems presented by the programmer's apprentice system. Some of the work on this system was started in earlier work by Rich and Shrobe which is attached as an appendix.

The essential feature of the system to be presented here is its relationship to a knowledge base with well defined semantic primitives. The deductive system is essentially a symbolic evaluator of abstract programs which are defined in terms of primitive actions corresponding to basic possible actions on data structures.

Since the basic actions of such a system correspond to higher level semantic notions, a trace of such actions meaningfully summarizes the dependencies between sub-modules of a program. These primitive traces also include reliance on assumptions in the knowledge base which, in turn, are marked for dependence on design decisions. Thus, the basic structures needed for program evolution are produced as a natural by product of the plan verification system's operation.

The Epistemology of REASON

REASON is primarily concerned with issues presented by side effects on complex data structures. Its most basic notion therefore, is that of part and generic part which name or index (respectively) the substructures of a data structure. Parts may be required to meet certain conditions, the simplest of which is that they must belong to a particular class of objects. For example, an alist might be described as having two parts: first and rest with the first being required to be of type dotted pair and the rest required to be either an alist or nil.

Another useful notion is that of property such as the length of a list, etc. Properties of an object are defined in terms of its sub-structure. For example, the length of a list is defined to be 1 plus the length of the rest of the list; the length of nil is tautologically zero. This implies that changes to the sub-structure of an object will result in changes of its properties.

Given the notion of property it is possible to state stronger restrictions on the well-formedness of a data structure. For example, buckets in a CONNIVER type data base usually have two parts, a count and a membership list. The count is required to be equal to the length of the membership list; such a requirement is called a constraint.

Finally, REASON has a notion of relations between objects. A common example of a relation is membership. Relations like properties are defined in terms of the sub-structure of the objects being related. For example, membership in a list is defined recursively as follows: either the object is the first object of the list, or a member of the rest of the list; nothing is a member of the empty-list (nil). Membership in a hash-table is defined as follows: an entry is a member of a hash table if it is a member of the bucket hashed to by the key part of that entry.

Notions relating levels of description

The above notions are adequate for presenting a conceptual description of an object. They do not, however, deal with questions of implementation, i.e. of building objects of one conceptual type by a pattern of interactions between objects of simpler types. As we discussed earlier this is a crucial task in any engineering project. For example, let us return to our earlier example of implementing a stack using an array and a cell.

The first thing we would have to state is the set of objects being used in the implementation scheme; thus, the array and the cell are implementation-parts of the stack. A second type of information is an implementation-mapping of conceptual objects onto implementation-parts. For example, the statement that the top of the stack is the array-item pointed to by the contents of the cell is exactly such a mapping. Such mappings bear many similarities to relation definitions in that they are both equivalences. Furthermore, the mapping specifies how side-effects to implementation-parts will result in side-effects to conceptual objects in a manner quite similar to that in which relation definitions describe how changes to part structure result in changes to relationships between objects.

The above propagation of effect from implementation to conceptual object has an important additional implication. The input output specification of segment behavior is an intrinsic description; it specifies what the module does independent of how that behavior interacts with that of other segments. But, the introduction of implementation mappings allows already existing program segments to take on extrinsic meanings. For example,

suppose that a CONNIVER bucket is implemented as a dotted pair whose left half is the count and whose right half is the list of members. In this situation, CAR has an extrinsic meaning of extracting the count from a bucket (in contrast to its intrinsic description of extracting the left half of a cons).

A final notion which must be added is that of virtual objects. Frequently one will imagine or conceptualize the existence of an object which has no existence as a unified whole within the program. The fringe of a tree is such an object. Another example is a subarray between two bounds of another array. Such virtual objects are side-effected by side effects to the concrete objects from which they are created. If, for example, a tree has its left branch changed, this will usually result in a change in the fringe. Conversely, if one is told that the fringe has been changed by certain actions, one can infer a range of possible actions to the tree itself which might have caused the stated changes.

This would be stated by a <u>virtual-object-description</u> which maps the part structure, properties and relations of the virtual object onto those of the concrete object from which it is derived. Such descriptions are quite similar in form and use to relation-definitions and implementation mappings.

Basic Actions In REASON.

REASON is structured as a symbolic program evaluator, i.e. it acts as if it were an interpreter running a program on symbolic (i.e. typical rather than actual) arguments. The programs are specified not as code, but as data flow between program segments which are in turn specified by input-output specifications. In the simplest case, REASON is given such an input-output specification for a main segment and for the sub-segments which will be used to achieve the behavior specified by the main segment. In addition, the data flow connections between these sub-segments will be given. REASON will symbolically evaluate this program using a relational data-base organized into time snapshots called <u>situations</u>.

In the initial situation REASON asserts the input conditions of the main segment. Then for each sub-segment it proves that the input conditions are satisfied. If this proof is successful, a new situation is created and the output assertions of the sub-segment are added to this new situation. The next sub-segment is then treated similarly. When all sub-segments have been evaluated, a proof of the output conditions of the main segment is attempted. If this succeeds, the plan at this level is correct. If any of the proofs fail, the plan has a conceptual bug and debugging intervention is required.

The result of this action is a tree of situations and a record of every deduction used in the proofs required for the plan verification. This network of connections between assertions is a verified plan of the main segment.

Output assertions of a sub-segment fall into one of three categories: a) New information b) Referent resolution c) Side effect. In the first case processing is relatively simple. The information is added to the data base and antecedent inferencing is performed. In particular, all relevant type information, constraints, relation, implementation, and virtual object definitions are used to infer new information. For example, on learning that ENTRY-1 is a member of LIST-2. demons are fired which check if LIST-2 is a sub-list of any other lists. If so, the inference is drawn that ENTRY-1 is also a member of those lists.

Frequently in making an assertion it is necessary to refer to an object by its relationship to other objects, for example "the car of list-1". Referent resolution involves searching the data-base for information which would identify such an object. If no such object is already known, then a new anonymous object is created which is asserted to meet the requirements of the reference.

The main difficulty, however, is in processing assertions specifying side effects. To illustrate this, suppose that it is known that ENTRY-1 is a member of a hash-table TABLE-1, but it is not known what the key-part of ENTRY-1 is. Suppose BUCKET-5 is a bucket of the TABLE-1 and that it is changed so that no entries with KEY-10 are members of it any longer. It is then possible that ENTRY-1 was deleted from the TABLE-1 (i.e. since its key might have been KEY-10), but it also possible that it wasn't. The data-base must be update in the output situation such that nothing which might be false is asserted to be true, and demons must be created to wait for and to propagate the necessary unknown information. Such processing can be done by using the chains of potential dependence given by relation definitions. Rich & Shrobe describes this process in detail.

A similar difficulty is brought about in the propagation of side effects to implemented and virtual objects which depend on the effected object. For example, if a queue is implemented using an array and two cells, then the efect of changing the value of one of the cells has to be propagated across the conceptual boundary so as to specify the changed state of the queue which is being implemented by the cells and the array.

Conversely, were a side effect specified only in higher level terms, the data base would need to be updated to propagate the knowledge to more primitive levels of description. For

example, using the above scheme of an array and two cells for a queue, if we were told that a POP had been performed we should make the appropriate changes to the cell pointing at the top. Sometimes the relationship between levels is less direct and results not in the propagation of new facts, but rather in a range of uncertainty.

The Specification Language

REASON provides a specification language for segment behavior whose semantics is given by the above primitive actions. This language is presented in detail in Rich & Shrobe. A program's specification is given by four clauses: inputs, outputs, expect, and assert. The first two of these merely provide internal names for the objects which are the inputs and outpus of the segment. The expect clause gives a list of conditions which the segment requires to be true of its input objects at the time the segment is applied. The output clause lists those conditions which the segment guarantees to be true on exit.

This last clause, however, has some added feature. In particular, it allows pseudo-statements of the following form: a) (NEW OBJECT-1) which specifies that OBJECT-1 is newly created during the execution of this segment; b) (ID OBJECT-1 OBJECT-2) which specifies that OBJECT-1 and OBJECT-2 name the same object and that this object has undergone a side-effect during the execution of this segment. One of these names is an output name, the other an input name. By using the output name one can make the clause be applied in the output situation, and similarly for the input name. Thus, in the following:

```
(specs-for reverse-by-side-effect
          (inputs:  list-1)
          (expect:  (list list-1))
          (outputs:  list-2)
          (assert:  (id list-2 list-1)
                    (list list-2)
                    (reverse list-1 list-2)))
```

we concisely specify that the cell named by list-2 on exit from reverse-by-side-effect contains a list which is the reverse of the list which was contained in that same cell at the time of entrance to this segment (N.B. this is not quite the nreverse of MacLisp).

Finally, a bracket notation is used to indicate reference. Thus [FIRST LIST-1] is an

abbreviation for the object which is the first part of list-1. Combining this with the multiple name convention above allows a time-based notation. For example we can easily specify that a cons has been reversed by side-effect as follows:

```
(specs-for swap
 (inputs: cons-1)
 (expect: (cons cons-1))
 (outputs: cons-2)
 (assert: (cons cons-2)
          (id cons-2 cons-1)
          (left cons-2 [right cons-1])
          (right cons-2 [left cons-1])))
```

where in resolving the meaning of the brackets the use of input or output names guides the choice of situation in which to resolve the reference.

Statements in this specification language can be translated to programs written in the language of the primitive actions described above. Thus, REASON can act on such specifications either by prior translation and direct execution or by interpretation.

What We Propose To Do

The central issues which distinguish our work from other work in the area of program verification is in the relationships we see between verification, plans, and the tasks of program maintenance and design. Specifically, we see the verification system as leaving behind a plan which is a trace of its semantic actions. This plan can be used in analyzing the effects of proposed pertubations of the program. Furthermore, such plans can be recorded in a library; the more abstract the plan is, the more it will serve as a prototype which a design system could draw upon.

REASON as it now exists was a throw away implementation used to investigate the necessary primitives needed for reasoning about the behavior of programs with side effects on complex data structures. It has been a useful tool, but the time has come to clean up its structure so as to allow flexibility in the uses outlined above. Thus, our first avenue of investigation is a clean specification of the behavior now exhibited by REASON and a clean implementation to go with it.

The second main avenue of exploration will be in using plans produced by REASON to examine the implications of proposed modifications and additions to a program. In particular, we will look at examples such as that given above of changing a program to reflect changes in a design choice. Such investigations will attempt to use the knowledge base of the system to the greatest degree possible.

A similar process used in design will also be investigated, namely the further specialization of already existing library plans. This will be used as an aid in interactive design, where the user might say something like "search the list computing a running total and exit when the item with key-5 is found." Such processing would involve specializing a search loop to also be an accumulation loop.

Finally various issues dealing with the control of proof processes will be investigated, in particular: the use of prototype plans to guide the proof of a specialization of that plan, the use of knowledge about data-structures to guide case-splitting, and the use of explicit recording of dependency and sub-goaling information to guide search <DeKleer et. al., 1977>.

Our methodology will involve coding a small, but realistic data-base oriented system which uses a variety of data structures such as hash-tables, CONNIVER-like tables, queues, stacks, arrays, records, etc. We will attempt to be specific at each stage about our design choices and modularization. Our intention is to pick an appropriate implementation stage, attempt to get REASON to understand it as fully as possible, and then to add new features, using REASON in evolution mode. The program might be something like a program to keep track of financial networks such as interlocking boards of directors of large corporations and banks, nations and markets of control, etc. It is felt that this program has extrinsic worth other than as a toy for REASON to play with and that therefore it will provide realistic (but not frantically changing) material for investigation.

Relationship to Other Work

There are four closely related characteristics of our work which separate it from other work on program verification. First, we regard verification as being factored into two stages the development of a verified plan at some level of abstraction and then after coding has been done, the <u>recogntion</u> of the code as a valid implementation of the plan. Thus, we see the verification process as being concerned with abstraction and implementation layering, rather than with the programming language itself. Second, we have made the issues raised by side effects a primary concern of our design. Third, we regard verification as being a means as well as an end. The verification process is the means to the recording of a plan structure which shows the dependencies between modules and design choices. This structure is the key to program design and evolution. Finally, our work is very much concerned with the structure, use and semantics of a programming knowledge base which will be used to guide verification and to capture important generalizations.

The Weakness Of Traditional Floyd-Hoare Logics

In our view, a program is a conceptual structure which at any point in its development can be viewed as having a heirarchy of abstractions. Each level can be regarded as being a program in the language comprised by the modules of the next lower level <Djikstra, 1976>.

Floyd-Hoare <Floyd, 1967> <Hoare, 1969,1971> logics tend to attribute primacy within this heirarchy to the programming language itself. Thus, the basic Floyd-Hoare approach

defines the language by attaching axioms to each primitive of the programming language. Two general methods are used <Igarashi, London & Luckham, 1973; King, 1969; Deutsch, 1973>, the difference being direction of action. In one the axiom specifies how to move a logical condition forward over a program primitive; in the other method the axioms specify backward motion.

Thus starting with the incoming predicates of a program, one can move statement by statement through the program, generating new logical sentences called verification conditions until one arrives at the other end of the code. The conjunction of the final verification condition and the output predicate of the program is a predicate equivalent to the statement that the program is correct. (In the other method, the same thing is done in the other direction.) This logical statement can then be handed to a theorem prover of any sort. If the statement can be proven, then the program is correct.

Given this bias toward the primitives of the programming language, Floyd-Hoare based systems tend to have inadequate concepts of implementation level or of interdependecy between such levels or of interdependency between modules at a single level of abstraction.

A second central problem of Floyd-Hoare systems is that they are oriented only towards verification, that is towards demonstrating that a program meets its specification. Such systems tend not to be oriented towards the explanation of how such a system meets its specifications or of isolating the bug responsible for its not meeting its specifications. In our view this is due to two factors: first, semantics are given solely by the axiomatic definitions of the primitives of the programming language (which does not allow enough abstaction), second these systems rely on uniform logic systems such as resolution theorem provers.

The practical consequence of these decisions is that Floyd-Hoare systems have no easy way of recording the semantic dependencies in the program. Lacking such a recorded network of semantic dependencies such systems cannot provide the support necessary to index a completed design for use in future modification or to identify the source of a program bug.

Floyd-Hoare systems have another major failing which arises from a similar source, namely that complex data-stuctures with side effects appear to be beyond the scope of most such systems. Since the design decision of F-H systems is to specify everything in terms of axioms describing the behavior of the programming language primitives, this requires the

axiom for RPLACA (for example) to specify all the possible consequences of changing the left half of a CONS cell. Given that the cons could be implementing any of an infinity of conceptual stuctures at a higher level, this puts an overly heavy burden on the description of the behavior of such program primitives.

Within the context of Floyd-Hoare logic, only Suzuki <Suzuki, 1976> has addressed the question of side-effects at all. In his system, a user has to provide reduction rules for the behavior of objects with side effects. These rules tend toward the ad hoc and in any event are difficult to write and are error prone. Further, his system still lacks the ability to make meaningful recordings of its action or to reason at an abstract level.

SIMULA-Like Systems

One of the main drawbacks of Floyd-Hoare systems is the failure to address the issues raised by implementation heirarchies. Simula like systems such as CLU <Liskov 1974; Liskov & Zilles, 1975> and ALPHARD <Wulf, 1974> have attempted to deal with these questions by adding to the language methods of accumulating modules which together constitute the behavioral repertoire of a conceptual object. A verification methodology is then worked out which first verifies that these modules do implement the conceptual behavior desired. Then, outside the cluster (or form in ALPHARD) only the conceptual behavior may be referenced.

This approach is clearly quite similar to our own structuring of the knowledge base around data-types. However, the systems still use Floyd-Hoare like methodology within each module and so in our view are overly restricted from dealing with abstractions, side-effects and pertubation analysis.

ACTOR Oriented Systems

A further step away from Floyd-Hoare type systems is taken in the work of Hewitt and Smith <Hewitt & Smith, 1975> and Yonezawa <Yonezawa 1975, 1976a, 1976b> who are working within the ACTOR formalism of computation. There has been considerable cross fertilization between this work and our own. Both projects have broken from Floyd-Hoare logic using situational data-bases and forward symbolic evaluation. Both projects have been concerned with side-effects from the outset.

There have been differences in emphasis, however. Our work has been geared towards

the design of the appropriate small set of descriptive primitives for the knowledge base and the actual implementation of a working system. Their's has been geared towards the development of the ACTOR formalism and towards a new language, PLASMA, designed to reflect the ACTOR viewpoint. In addition, their work does not reflect our commitment to a factorization of the process into plan verification and code recognition.

The Programmer's Apprentice Project

Our work was begun as a joint venture with Charles Rich. The result of that work is reported in our joint Master's Thesis and a subsequent Technical Report. Our joint work is now continuing by each of us deeply pursuing a single aspect of the project. Rich is centering on plan recognition and the design and structure of the programming knowledge base for the joint system <Rich, 1977>. In addition, Richard Waters <Waters, 1976> is working on a similar project using numerically oriented Fortran programs as his domain of reference.

These projects will cross fertilize one another in several ways. Our plan verification system will produce plans for Rich's system to use in recognition; his work will clarify structures of the knowledge base which will lead to useful guidance for our system, particularly in pertubation and design. Ultimately, these separate avenues of exploration will be brought back together in a full blown programming environment incorporating and synthesizing all of the above work.

Other Work on Reasoning and Analysis of Engineered Devices

Much of our work on reasoning bears a resemblance to work being done by the Engineering Problem Solving group at MIT <Sussman, 1977>. In particular our reasoning system uses techniques quite similar to Analysis by Propagation of Constraints <Stallman & Sussman, 1976>. Indeed, our evolving notion of the structure of REASON has been influenced substantially by AMORD <DeKleer et. al., 1977>. However, our work has been considerably more concerned with reasoning about the dynamic behavior of systems (side-effects) than has the above work. Similarly our work on reasoning shares many common features with Moore's system <Moore, 1975> although we have been specialized in our concern for programs as a problem domain. As our work moves into the border-line area of design we expect to be guided somewhat by McDermott's <McDermott, 1976> work on the design of electronic circuits.

The Psi System of Barstow,Kant and Green

Finally, many of ideas for the codification of knowledge in a data base will be guided somewhat by the work of Barstow and Green <Green & Barstow, 1975> who as part of their PSI system have developed a production system for program design. Barstow's part of the system proposes a tree of designs to meet a conceptual description of the program requirements which is pruned for efficiency reasons by Kant's <Kant, 1977> effiency expert. Barstow's production system in our view represents a very useful refinement heirarchy for thinking about programming concepts. However, his system does not seem to contain any deep teleological notion other than that given by the refinement path itself. Thus, in our view his system will be limited in its ability to conduct complicated design or pertubation since these tasks involve interactions between synthesis and analysis, thus requiring explicit teleological structure. However, we believe that a wedding of the concepts in PSI to our own will be highly productive.

Bibliography

Boyer, R.S. & Moore 1975, J.S. Proving Theorms About LISP Functions, JACM vol. 22 no. 1 , January 1975.

Deutsch, L.P. 1973, An Interactive Program Verifier, PhD. Thesis University of California at Berkeley, June 1973.

Dijkstra, E.W. 1976, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J. 1976

DeKleer, J., Doyle, J., Steele, G. & Sussman, G.J. AMORD: Explicit Control of Reasoning, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, August 1977.

Floyd, R.H. 1967, Assigning Meaning to Programs, Mathematical Aspects of Computer Science J.T. Schwartz (ed.) Vol. 19 Am. Math. Soc. Providence R.I. 1967.

Green, G.C. & Barstow, D.R. Some Rules for the Automatic Synthesis of Programs, IJCAI-4 Tbilisi, USSR, September 1975.

Hewitt, C. & Smith, B.C. 1975, Towards A Programming Apprentice, IEEE Transactions on Software Engineering, Vol. SE-1 No. 1, March 1975.

Hoare, C.A.R 1969, An Axiomatic Basis for Computer Programming, Comm. ACM, vol. 12, number 10, October 1969, pp. 576-580,583.

Hoare, C.A.R 1971, Proof of A Program: Find, Comm. ACM, vol. 14, number 1, January 1971, pp. 39-45.

Igarashi S., London R., and Luckham D. 1973, Automatic Program Verification I: A Logical Basis and Its Implementation, Stanford AIM-200, May 1973.

Kant, E. The Selection of Efficient Implementations for A High Level Language, Proceedings of the Symposium on Artificial Intelligence and Programming

Languages, August 1977.

King, J. 1969, A Program Verifier, Carnegie Mellon University, 1969.

Liskov, B. 1974, A Note on CLU, MIT/Computation Structures Group Memo 112, MIT/LCS, November 1974.

Liskov, B. & Zilles, S.N. 1975, Specification Techniques for Data Abstractions, IEEE Transactions on Software Engineering, Vol. SE-1 No. 1, March 1975.

McDermott, Drew Vincent 1976, Flexibility and Efficiency in a Computer Program for Designing Circuits, MIT PhD. Thesis, September 1976.

Moore, Robert Carter 1975, Reasoning From Incomplete Knowledge In A Procedural Deduction System, MIT/AI-TR-347 December 1975.

Pratt, V. 1976, Semantical Considerations on Floyd-Hoare Logic, MIT/LCS/TR-168, Septermber 1976.

Rich, C. 1977, Plan Recognition In A Programmer's Apprentice, MIT/AI Working Paper 147, May 1977.

Rich C. and Shrobe H. 1976, An Initial Report On A LISP Programmer's Apprentice, MIT/AI/TR-354, December 1976.

Spitzen, J. & Wegbreit, B. 1975, The Verification and Synthesis of Data Structures, Acta Informatica 4, 1975.

Sussman, G.J. 1977, The Engineering Problem Solving Project, Research Proposal Submitted to the National Science Foundation, MIT/AI July, 1976.

Suzuki, N. 1976, Automatic Verification of Programs with Complex Data Structures, Stanford AIM-279, February 1976.

Waters, R.C. 1976, A System for Understanding Mathematical FORTRAN Programs,

MIT/AI Memo 368, August 1976.

Wegbreit, B. 1976, Constructive Methods In Program Verification, Xerox Palo Alto Research Center CSL-76-2, July 1976.

Wulf, W.A. 1974, ALPHARD: Towards a Language to Support Structured Programming, Carengie Mellon University Dept. of Comp. Sci., April 1974.

Yonezawa, A. 1975, Meta-Evaluation of Actors With Side Effects, MIT/AI Working Paper 101, June 1975.

Yonezawa, A. 1976a, Symbolic-Evaluation As An Aid To Program Synthesis, MIT/AI Working Paper 124, April 1976.

Yonezawa, A. 1976b, Symbolic Evaluation Using Conceptual Representations For Programs With Side-Effects, MIT/AI Memo 399, December 1976.

Zilles, S. 1975, Abstract Specification for Data Types, IBM Research Laboratory, San Jose California, 1975.