Massachusetts Institute of Technology

Artificial Intelligence Laboratory

Cambridge, Massachusetts

A.I. Memo 368

May 1976 revised August 1976

A System For Understanding Mathematical FORTRAN Programs

by

Richard C. Waters

ABSTRACT

This paper proposes a system which, when implemented, will be able to understand mathematical FORTRAN programs such as those in the IBM Scientific Subroutine Package. The system takes, as input, a program and annotation of the program. In order to understand the program, the system develops a "plan" for it. The "plan" specifies the purpose of each feature of the program, and how these features cooperate in order to create the behavior exhibited by the program. The system can use its understanding of the program to answer questions about it including questions about the ramifications of a proposed modification. It is also able to aid in debugging the program by detecting errors in it, and by locating the features of the program which are responsible for an error. The system should be of significant assistance to a person who is writing a program.

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 Project Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

ACKNOWLEDGMENTS

I would particularly like to acknowledge the assistance of C. Rich with whom I discussed many aspects of this paper. In addition I would like to thank G. J. Sussman, B. Smith and A. L. Brown for their comments.

CONTENTS

CONTENTS

I. A PROGRAM UNDERSTANDING SYSTEM

This research project is concerned with designing and implementing a program understanding system. This section briefly describes what the proposed system does, how it does it, and why it is worth doing. Sections II and III specify, in greater detail, what the system does and how it does it. This paper speaks of the program understanding system in the present tense. However, it should be noted that the system has not been implemented yet.

I.1 BRIEFLY, WHAT THE SYSTEM DOES

The system understands mathematical FORTRAN programs. It does not attempt to understand the mathematics embodied in a program, but only the programming. A mathematical program can be considered as the implementation of a theorem. The system does not try to understand the theorem. It just believes it. What it does do, is to understand how the program implements the theorem.

Mathematical FORTRAN programs were chosen as the domain because they are a straightforward type of program. They are real programs that use only a small subset of possible programming techniques. In particular, they use only simple data types (numbers, arrays, functions), simple control structure (no recursion, no asynchrony), static variables and no I/O. Furthermore, in the IBM SSP subroutine library, there are a large number of reasonably structured real programs to serve as experimental data for the system. These programs are a good test of a program understanding system, because they were not specifically written to be understood by such a system. The system demonstrates its understanding of a program through its ability to perform several tasks which require understanding.

I.1.1 ANSWERING QUESTIONS ABOUT A PROGRAM

The system is able to answer questions about a program it understands, such as:

a) What is this part of the program?
b) What does this part do?
c) Why is this part here?
d) What is the function of this part?
e) How does this part do what it does?
f) What part achieves this goal?

In other words, it is able to explain a program, to impart its understanding of it to another.

I.1.2 DETECTING INCONSISTENCES IN THE PROGRAM AND ITS DESCRIPTION

The system does not attempt to prove the correctness of a program. However, it is able to detect simple inconsistences in its understanding of a program. It can detect a variety of problems where it can be simply shown that a segment of a program can not possibly achieve the results requested of it. The system's deductive apparatus consists largely of pattern matching, and trial by example. This allows it to prove many assertions false, but few correct.

I.1.3 AIDING IN THE DETECTION AND UNDERSTANDING OF BUGS

The system can apply its understanding of a program to aid in the task of debugging it. When running the program in a careful mode, the system constantly checks whether a contradiction has arisen between its understanding of the program, and what is actually happening. The moment a contradiction appears, the system reports it. This causes bugs to be found closer to their point of origin than in an ordinary programming system. For example, the system might say "matrix A is not in hermetion normal form" rather than "zerodivide" forty subroutine calls later.

Further, once given a point of departure, the system can trace back even closer to the origin of the problem. For instance, after discovering that matrix A was not in hermetion normal form, the system might say "the subroutine F is not living up to its extrinsic description, which claims that its

output is always in hermetion normal form" or "the theorem implied by the use of subroutine F that any matrix with properties P1, P2, and P3 is in hermetion normal form, must be false."

In short, the system assists a user in backtracking a bug to its source, and by watching the execution closely, reduces the amount of backtracking which needs to be done.

I.1.4 INVESTIGATING THE RAMIFICATIONS OF A MODIFICATION

When a segment of code is altered, to fix a bug or add a feature, the system can assess some of the ramifications of the change. This is particularly useful when the changed segment served other purposes in addition to the one under consideration. The system can ask itself whether the functions the old segment served are still being taken care of, whether the new goals interfere with other goals of the program, and whether the change does in fact achieve the results intended of it.

I.2 BRIEFLY, HOW THE SYSTEM WORKS

The system looks at a program as being composed of logically separate segments of code which are combined together by connective tissue. Further, each segment is composed of subsegments, etc. The connective tissue is of two types, data flow connective tissue (variables, function parameters, assignments), and flow of control connective tissue (branches, subroutine calls, sequential code placement). The data flow connective tissue transmits data between segments, and the control flow connective tissue executes the segments in the proper order.

I.2.1 THE STRUCTURES WHICH DESCRIBE A PROGRAM

The program as a whole is described through the interaction of two types of descriptions: behavioral descriptions and plans. A behavioral description specifies what a segment of code does without indicating how it is done. It lists the inputs, outputs, prerequisites, and output assertions of a segment. Behavioral descriptions describe a segment from two major points of view. An intrinsic behavioral description tells what a segment does in isolation. All the statements it makes about a segment are true for every use of the segment. An extrinsic behavioral description tells what a use of a segment does in the context of its use. One segment can be used for many logically unrelated tasks in the same program.

A plan indicates how several segments (and their extrinsic behavioral descriptions) combine to form a larger segment (and its intrinsic behavioral description). Plans are quite variable, but, observation indicates that they fall into a small number of types (around ten). This makes it possible to deal with the plans even though each type is treated separately. It also means that a great deal of information can be inferred about a segment purely from the type of the plan for the segment, since out of the vast array of possible plan types only a small number are used. This in turn makes both recognizing and understanding a segment easier.

The grand plan, which completely describes the operation of a program, simply consists of behavioral descriptions and plans for every segment down to some level where the segments are taken as fundamental and as having no subsegments and hence no plans.

I.2.2 HOW THE SYSTEM USES THE DESCRIPTIVE STRUCTURES

Questions about a segment are answered through reference to the descriptive structures. For example, "What does this do?" is answered by reference to the behavioral description. "How does it do it?" is answered by reference to the plan. "What function does it serve?" is answered by reference to the extrinsic behavioral description and the plans in which it is contained.

Detection of inconsistences is performed while the descriptions are being constructed. As the system builds up its understanding of a program, it continually checks for inconsistences in what it knows. Whenever it discovers or is told something about the program, it attempts to verify it. If it discovers a contradiction, it reports it. If it verifies it, fine. Most of the time however, it comes to no conclusion, since the deductive mechanism is weak. In that case, it assumes that the fact is true, but

is prepared to discover at a later time that the fact is contradicted

More complex tasks are performed by means of the system asking itself questions. For instance, when a bug is detetected the system asks questions such as, "Where did this come from?"; "Who wanted it that way?"; and "Who wants it to be another way?" The answers to these questions lead to an understanding of the bug.

I.2.3 HOW THE DESCRIPTIVE STRUCTURES ARE BUILT UP

The system develops its understanding of a program based on the code itself, and on what the user tells it about the program via comments. Looking at the code, the system can separate out most of the control and data flow connective tissue. Also, with a knowledge of what the primitive units are, the system can go a long way toward analyzing the lower level segments.

The user must provide comments describing the overall behavioral description of the program, and the basic segmentation of the code. Most comments are in the form of either a partial specification of a behavioral description, or the delineation of a segment combined with an indication of the plan type applicable to it.

It should be noted that it is very difficult for the system to determine the segmentation by itself, because of the vast number of possible segments it must consider, and because there are several transformations commonly applied to segments which improve program performance at the expense of clarity. For example, code is shared or similar subsegments are factored out of logically unrelated segments. Once the system has a handle on the segmentation of the program it can use its knowledge of the stereotyped plan segment types to analyze the program further.

The amount of annotation which the user is required to make is a critical parameter. If too much annotation is needed the system will be too cumbersome for practical use. This not withstanding, the currrent goal of this research is to achieve understanding without excessive concern for how many comments are needed.

I.3 WHY THE SYSTEM IS VALUABLE

The size and complexity of programs are rapidly increasing. This makes programs harder to work with and harder to understand. This type of system would be very useful for acquainting or reacquainting a person with a program, and for keeping track of what is going on when a person works on a program.

The system would be particularly useful in a situation where a group of people are working on a program. The system could keep each person abreast of what the others are doing. In addition, it could help coordinate what the people were doing by watching that the segments people were writing would interface properly, and that the goals would mesh together.

In its full form, the system would be an aid in debugging. Bugs seem to be of two types: errors in the algorithm, and errors in the implementation of the algorithm. The system would be helpful in locating and understanding implementation bugs in particular. Many bugs are simply due to forgetting minor details which, though trivial, are essential. The system would also be able to greatly reduce the chance of a programmer producing a new bug while fixing an old one. This is usually due to forgetting that a particular segment of code has more than one function in the program.

In the future, the system ought to be able to move closer to a program verification system. A true understanding of a program should lead to a proof of correctness. The understanding serves as a plan for the proof.

In a similar fashion, the system leads toward automatic programming. A complete understanding of a program should enable the system to write the program, following the plans in its understanding. The only question is how can the understanding be developed without reference to the code.

I.4 RELATIONSHIP TO OTHER WORK

This section describes the relationship between the system proposed in this paper and other work, from two points of view. First, it compares the basic methodology of this system with other

approaches to the same overall goal. Second, it compares this system with other systems using a similar methodology. While making these comparisons, it tries to trace the development of the key ideas embodied in this system.

I.4.1 OTHER APPROACHES TO THE SAME GOAL

Consider a user (programmer) who wishes to perform a certain computation. To do this, he writes a program which will cause a digital computer to perform the computation. This reduces his work to the work required to write the program. The basic goal of the research proposed in this paper is to reduce programming effort as much as possible.

Writing a program consists of two intertwined tasks, designing all of the details required to cause a computer to perform a computation, and showing at least informally, that this computation is the one the user had in mind. Designing the details (producing the program) is what is usually referred to as writing the program. However, it is clear that a programmer is continually guided by an informal feeling for why what he has written, and what he will write, is correct.

I.4.1.1 GENERAL PURPOSE HIGH LEVEL LANGUAGES

The first step in reducing programming effort was achieved by moving from machine code, through assembler language, to general purpose high level languages. The development of these more powerful languages stemmed from two key ideas: modules and abstractions.

A pre-written module can be as simple as a multiplication routine or as complex as a data base management system. A module can be used as a subroutine or expanded inline as a macro. It can be partially pre-evaluated or transformed after instantiation to increase efficiency. In any case, modules reduce the effort required to write a program because they can be used without having to be rewritten. They reduce the effort required to verify a program because they can be used as lemmas in the verification without having to be reverified.

Abstractions are used to express certain key relationships in a program in ways which are more convenient for the programmer. For instance, assembler languages introduced, among other things, the idea of using symbols as variables to indicate data flow. Compiled languages introduced, among other things, the idea of using syntactic nesting to indicate data flow. They also introduced the idea that all of the primitives of a language could be modules, freeing the language from any resemblance to, or dependence on, any particular computer architecture.

Global operations (performed by an assembler, compiler, or interpreter) translate these abstractions into machine understandable form. This frees the programmer from specifying details, such as actual memory addresses, which, though they must eventually be determained in order for the computation to be performed, are not of any direct interest to the user.

The abstractions facilitate verification because they correspond more directly to the properties needed for verification. For example, verification may require that data flow occur from a use of module A to a use of module B. The syntactic nesting of the use of module A in the argument list of the use of argument B succinctly expresses this requirement. Any added mechanism of memory addresses, stacks, parameter passing, or variables would just get in the way.

General purpose high level languages (such as FORTRAN, PL/I, and ALGOL) have introduced a large number of features of general usefulness. They have introduced abstractions such as subroutines and data types, and modules such as access functions for complex data types (like strings, arrays, and structures). These languages have greatly simplified the programming process while maintaining general applicability by incorporating knowledge of programming techniques. However, they have not gone far enough; programming is still difficult.

I.4.1.2 SPECIAL PURPOSE SYSTEMS

A number of systems have been developed which are more powerful than general purpose languages due to the fact that they incorporate algorithmic information specific to particular problem domains. The simplest of these systems are subroutine packages (such as the IBM SSP [IBM

GH20-0205-4]) which extend the power of a general purpose language through the addition of domain specific modules.

To produce systems with greater power, designers have incorporated domain specific knowledge in ways other than modules. For example, consider the language BDL [Hammer, Howe, Kruskal & Wladawsky 1975] which can be used to write business data processing programs. BDL constrains programs to have one particular top level structure. This allows the system to automatically generate the code needed to implement this structure. Further, all the modules and abstractions used in BDL are specifically designed to fit into this particular top level structure in order to make the user's design and verification tasks easier. All this makes writing a program easier as long as the particular top level structure is appropriate.

This trend can continue with systems containing more and more information about smaller and smaller domains, until the trend culminates in customizers. A customizer can only produce a few programs, but it knows all of the details of each of these programs. The only problem the user faces is discovering whether the computation he desires can be implemented by one of the programs the customizer can produce. If it can, the user specifies the program he wants by filling out a questionnaire. Verification is simple because the questionnaire is directly posed in the terms the user is interested in.

The problem with these systems is that they are too specific. A programmer can only use such a system if there happens to be one which applies to his problem domain. If he changes domains, he will have to learn an entirely new system. Further, the more powerful one of these systems is, the more limited its applicability.

I.4.1.3 USING GOOD PROGRAMMING STYLE

The particular style which a programmer uses has a great effect on the ease with which he can write a program. Languages have been developed which encourage good programming style (for example structured programming), and make some types of bad style impossible. For example, the language CLU [Liskov 1974] is designed to make unstructured data access impossible. In CLU, a data item can only be accessed through the access functions defined for its data type. Therefore, it is impossible to use any ad hoc data manipulations which could complicate the verification process. This approach extends the idea of enhancing the power of a general purpose language by including programming knowledge in the system.

I.4.1.4 AUTOMATIC PROGRAMMING

With any of the general purpose languages discussed above, a programmer faces the task of specifying what modules are to be used, and how they are to be interconnected. Automatic programming attempts to automate this process. In automatic programming, the user's program is a specification of what is to be done, not how it is to be done. Systems have been designed using specification through examples of behavior (such as I/O pairs [Summers 1975; Shaw, Swartout, & Green 1975; Hardy 1975] and traces [Bauer 1975]), exact specifications in a predicate calculus like language [Manna & Waldinger 1975], or English language descriptions [Ruth 1976].

It is not true that automatic programming would eliminate all the effort of programming. Developing a precise description of a desired computation is difficult, whether or not the description is algorithmic. However, assumedly, a programmer has always had to develop some descriptive specification for a program he wishes to write, at least in his head. Therefore, writing and verifying a program, which is itself a specification, should be easier, as long as the type of descriptive mechanism used by the automatic programming system is sufficiently similar to the one used internally by the user.

It should be noted that automatic programming systems do not eliminate the need for domain specific algorithmic knowledge. Rather they attempt to provide a uniform method of access to this knowledge. Domain specific knowledge must be realized in the programs produced by an automatic programming system. Therefore, if the user is not going to specify it, it must be either known by the system beforehand, or reinvented by the system in response to a problem.

Unfortunately, it has not yet been possible to codify large areas of knowledge, though small areas have been codified (for example, Green and Barstow discuss the codification of knowledge about sorting programs [Green & Barstow 1975]). Further, the problem solving capability required to understand specifications and write a large variety of programs based on a manageably small stock of knowledge is also beyond the state of the art. As a result, none of the automatic programming systems proposed has been made to work well enough to be of any practical use. Those which have been implemented at all, only work in a very small or simple domain such as simple list manipulation.

The approaches described above have been directed toward developing an interpreter for a language (whether algorithmic or based on specifications) which is so powerful that a complex program can be trivially written and verified in the language. This is certainly a laudable goal, however, it does not seem likely that it will be achieved in the near future.

I.4.1.5 AUTOMATIC VERIFICATION

Attempts have been made to develop a system which will at least be able to verify that a program is correct. This would be particularly useful since true verification has been neglected by programmers in the past.

In order to make automatic verification posssible, the behavior of the primitive elements of programming languages had to be rigorously axiomatized. Further, proof rules had to be developed so that the axioms about the primitive elements could be combined into a proof of claims about a program as a whole. This work was started by FLoyd [Floyd 1967] and continued by Hoare [Hoare 1969; Hoare 1971].

In order to prove something about a program (such as the correctness of its specifications), the key step is deciding what subtheorems to attack. This is analogous to picking subtasks in the process of writing a program, and unfortunately does not appear to be any easier. In addition, current theorem provers are not able to prove theorems of any great difficulty.

Straightforward selection of subtheorems is only possible in straight line programs. If there is looping involved (via gotos, looping constructs, or recursion), then heuristics must be used in order to develop subtheorems which describe the action of the loops (for example [Wegbreit 1973]). Systems, such as that of Boyer and Moore [Boyer & Moore 1975; Moore 1974], which attempt to prove theorems about a program by looking just at the program, are only able to work with simple programs.

Systems, such as that of Waldinger and Levitt [Waldinger & Levitt 1974], where the user specifies loop assertions, still bog down on relatively simple programs due to the weakness of current theorem provers, and the problem of codifying and using enough domain specific knowledge.

I.4.1.6 PROGRAMMING ASSISTANT SYSTEMS

There are several main ideas behind assistant systems. One is a recognition of the fact that, in the absence of automatic programming, programs are not written in one pass. They are written bit by bit, and then modified and added to many times as errors are found and corrected, and the original specification for the program changes. This idea leads to a desire for systems which combine editors, compilers, interpreters, and debugging aids into one interacting unit. This combination would facilitate cycling between modes and the detection of errors.

A second idea is to have the assistant system keep track of a myriad of details, and help the programmer avoid pitfalls. This is a natural extension of the idea of designing a language so as to encourage good programming style. An assistant system would utilize knowledge of programming in general, and of the particular language which the programmer was using in order to catch many small errors which might otherwise lead to big problems. What makes this qualitatively different from the many checks done by current compilers is that the bookkeeping would exist across the entire process of developing a program, not just within a single compilation of a single subunit.

By themselves, these two ideas are not very exciting. They should be usefull but not spectacular extensions of current programming systems. In order to really assist a programmer, a system must enter into the programming process, either in design or verification. The main tenent of

the programming assistant approach is that this is possible, even though the programmer will have to perform the two key tasks which are currently beyond the capabilities of automatic systems. As described above, these two tasks (which may be closely related) are non-trivial problem solving and the encoding and utilization of large amounts of knowledge.

Basic descriptions of programming assistant systems were put forward by Floyd [Floyd 1971] and by Winograd [Winograd 1973], however they did not suggest how such a system could be realized.

I.4.1.7 PROGRAM UNDERSTANDING

In order to cooperate with a programmer who is doing the really complex design and verification, a system must understand what the programmer has done and is doing. How this understanding is to be achieved is the central interest of this paper. The key idea is that along with each program there is a plan. The plan links the specification, the program, the verification, and the design of the program. It tells the purpose of each element of the program. This basic idea originated with Sussman [Sussman 1973a] and Goldstein [Goldstein 1974].

From the point of view of this paper, developing a plan is the primary activity of programming. If the plan is known, the specifications, the program, the verification, and the design can be derived. If a system could develop a plan just from the specifications, it could do automatic programming. If a system could develop a plan just from the specifications and the program, then it could do automatic verification.

This paper proposes a system which can develop a plan based on a program, partial specifications for it, and comments on it. This system is an initial step towards a programming assistant system which would understand a plan as it was evolved by a programmer, and assist in verification and writing of the program.

The research proposed by this paper is mainly directed towards the question of how a system can find out the plan for a program. It also investigates how plans can be represented and utilized by a programming assistant system.

I.4.2 SIMILAR APPROACHES

This section discusses a number of systems involved with programming assistance, program understanding, and plans. It contrasts these systems with the system proposed in this paper. Consider the diagram below. It represents some of the features of the system proposed in this paper. Other systems will be compared with this system particularly with regard to how they develop an understanding and how they use an understanding.

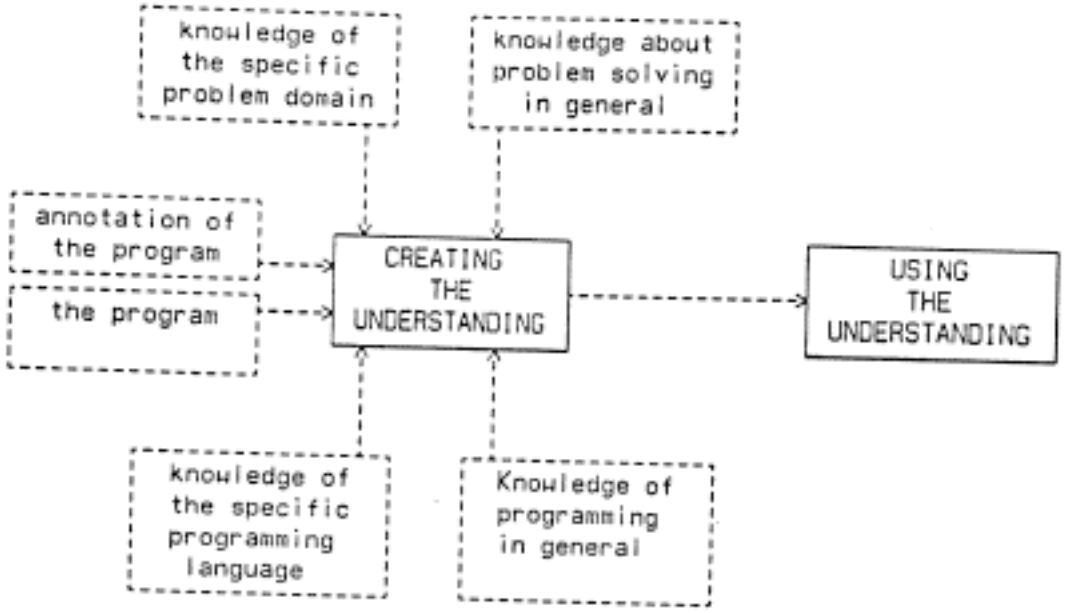


Fig 1: Diagram of data flow in a program understanding system

The system described in this paper takes a program together with annotation on it as input. Using general knowledge about programming and problem solving (which is embodied in the plan types), and knowledge about the specific programming languge (FORTRAN), it develops a plan for the program. This system does not attempt to encode much knowledge about the specific problem domain, because it is too complex, being the full range of applied mathematics. It uses the understanding it develops of a program in order to answer questions about the program, including questions about the ramifications of a modification. In addition, it can detect bugs in a program largely through an attempt to prove the program incorrect.

I.4.2.1 SUSSMAN

Sussman's HACKER [Sussman 1973a; Sussman 1974] is an automatic programming system which writes parameterless programs in the simple domain of the blocks world. It constructs programs given predicate calculus like specifications of the output state, and input state. It acquires greater skill in programming by learning from the process of writing each program it produces. HACKER writes programs by first proposing a simple-minded program and then debugging it. It alternately modifies the program and searches for bugs in it, until the program works correctly.

Each time a program is written, a plan for it is constructed. When a program is modified, its plan is modified to reflect the changes. The plan for a program represents what the purpose of each feature of the program is, and information about how the features interact. The plan is of central importance in detecting bugs and proposing solutions for them. HACKER does not have to attack the problem of recognizing the plan for a program because the parts of HACKER which write programs produce the plans at the same time.

I.4.2.2 GOLDSTEIN

Goldstein's MYCROFT [Goldstein 1974; Goldstein 1976] operates on programs in the simple domain of loop free turtle line drawing programs. These programs are output intensive programs similar to the command sequences sent to a plotter. Almost every component of a program can be identified with a visible segment of the resultant drawing.

MYCROFT accepts as input a program which may have bugs, and a partial description of the intended output. It then proceeds to construct a plan for the program. The user can optionally include commentary on the program indicating features of the plan.

Due to the specific nature of the domain, the task of finding the plan consists largely of deciding which parts of the actual output correspond to which parts of the desired output. MYCROFT attempts to find a plan which minimizes the number of errors detected. How easy this is to do depends on how well the natural segmentation of the program corresponds to the description of the intended output. This segmentation is a very important form of commentary supplied by the user.

MYCROFT uses the plan to propose fixes for the bugs it detects. It then proceeds to construct a corrected program. It is most successful when a bug is caused by an incorrect value (such as a line segment being too long). It has more difficulty when a bug is caused by incorrect logical structure of the program (such as pieces of the intended output completely left out). It seems reasonable that programming assistant systems in general will follow this pattern. This is because designing the basic logical structure of a program seems to be a more difficult task than picking the correct values to use.

I.4.2.3 RUTH

Ruth [Ruth 1974] developed a language which can be used to express the set of all algorithms for a given task. This allows him to encode the domain specific algorithmic information for a particular problem area. His system takes as input a program and the description of a set of algorithms. It then determines whether or not the program implements one of the algorithms. If a match is found, the corresponding algorithm is reported. If a match is not found, the system tries to find a near miss algorithm and reports differences from this as bugs. His system has successfully operated on real programs written by students learning programming.

His system performs a complex pattern matching task. It operates directly on the programs, which are written in a simple algebraic LISP like language having a conditional construct and a looping construct. Comments on the program are not used, except for the implicit comment that the program is intended to implement one of the specified algorithms. His system does a lot of work in order to deal with transformations which can be applied to programs. It does this in order to detect when two programs are essentially the same even though they appear, at first glance, to be very different. Transformations are also of considerable importance in the domain of FORTRAN programs. His system incorporates knowledge about common bugs so that near misses can be detected easier.

Ruth does not deal with plans as such. He identifies understanding with determining which algorithm corresponds to the given program. The algorithmic representation he uses captures some of the notions of a plan. However, it is more like a typical implementation of the algorithm, and does not contain teleological information about purposes.

I.4.2.4 BROWN

Brown's system WATSON [Brown 1975; Brown forthcoming; also: Sussman 1973b; Brown 1974; Sussman & Brown 1974], localizes failures in electronic circuits. As input, it takes a piece of electronic equipment (which it operates on by giving commands to a person who makes tests and adjustments), its circuit diagram, a plan for the circuit, and a description of the problem it is exhibiting. The plan and circuit are assumed to be bug free. The failure is therefore assumed to be due to a damaged component, or improper adjustment (note the similarity with the bugs which were easiest for Goldstein's MYCROFT to handle).

WATSON uses a variety of techniques triggered by features of the plan in order to localize the failure. The plans are constructed by hand and input to the system. As a result, WATSON does not attack the problem of developing an understanding of a circuit. However, his system shows the importance, and power, of knowing plans in the domain of electronic circuits.

I.4.2.5 GERHART

Gerhart's work [Gerhart 1975] is not involved with an automatic system. Rather, it introduces tools which can be used by a person who is writing a program, in order to make verification easier. She views programs as being built up out of simple pieces with known properties which are combined in ways which correspond to simple proof rules. This is in line with the basic work of FLoyd and Hoare. At an intermediate level between the basic elements and a program as a whole, she introduces "program schemata" which are the distillates of basic algorithms. She also introduces transformations which can be applied to a section of code without altering its behavior. These two things can be used as lemmas when trying to prove a program correct. They serve as an intermediate skeleton for understanding and justifying a program.

Gerhart does not address the issue of recognizing a program in terms of these schemata and transformations, or any way to automatically manipulate them. Her work is mentioned here because of the similarity between her program schemata and the plan types discussed in this paper. Further, the collection of schemata and transformations which apply to a program has many features in common with a plan for the program, even though there is no explicit teleological information.

I.4.2.6 HEWITT and SMITH

Hewitt and Smith's programming apprentice [Smith, Waters & Lieberman 1973; Smith & Hewitt 1974; Hewitt & Smith 1975] is a programming assistant system designed to work in the domain of Hewitt's ACTORS formalism. The major goal of their system is proving the correctness of programs.

In contrast to many other approaches, major emphasis is placed on comments. The central comment is the "contract." Contracts correspond to what are called behavioral descriptions in this paper and specify the input/output behavior of a subroutine. The ACTORS formalism encourages heavy subroutinization. Since each subroutine has a contract associated with it, this leads to a reasonable large amount of commentation.

In order to prove correctness, "meta-evaluation" (which is similar to symbolic evaluation), and "ACTOR induction" are used. Each subroutine is proved to satisfy its contract which is used as a lemma when proving the correctness of other subroutines.

Their system does not develop any descriptive structure in addition to the code and comments supplied by the user. The information corresponding to a plan is fragmented, and most of it is only present implicitly. Their system uses a form of comment called a "plan," however, it is different from the plans used in this paper. It is used to capture domain specific information and to guide the theorem prover which works with the meta-evaluator.

The system does not deal with automatic generation of descriptive structures. Any plan-like structures which it uses are entered by the programmer. However, the system does embody several ideas which are an important part of the research proposed in this paper. In particular it uses the idea of behavioral descriptions ("contracts"). Their work is being extended by Yonesawa [Yonezawa 1976]. However, meta-evaluation is still largely at the hand simulation stage.

I.4.2.7 GREEN and BARSTOW

Researchers at Stanford University have been doing considerable work on what they call "program understanding" [Green et. al. 1974]. However, from the point of view of this paper, their work would be described as directed towards automatic programming. Green and Barstow propose a program writing system which knows about simple sort programs [Green & Barstow 1975].

The user of their system would only participate by making high level design decisions. The system would have extensive knowledge of how to actually write sorting programs. Their paper concentrates on the informal reasoning which the system would follow, and on what knowledge the system would have to have. It does not discuss, in any detail, how the system could be implemented.

I.4.2.8 RICH and SHROBE

Rich and Shrobe's programming apprentice [Rich & Shrobe 1974; Rich & Shrobe 1976] is designed to work in the domain of LISP programs. It comes closer to the kind of programming assistant system discussed in this paper than any other system currently proposed. They touch on the full range of behavior of a programming assistant system, but concentrate on understanding data types, understanding programs through plans, and verification.

They recognize a hierarchy of plans of increased detail as the fundamental description of a program. They envision a scenario in which the plans are developed through a dialog with the user as a program is being written. However, they also investigate the question of developing the plan for a program written outside the system. They use plans primarily to guide verification which proceeds in a similar manner to Hewitt's meta-evaluation.

They have worked extensively with the example of the access functions to a hash table. They use knowledge about the data types, and algorithms in this domain in order to build the plan and aid in the verification. They have implemented some parts of their system, and are continuing implementation efforts at the current time.

The work presented in this paper has benefitted greatly from the ideas in Rich and Shrobe's work. This paper could be looked at as focussing on one feature of a programming assistant system, very similar to theirs, but working in a different domain.

I.4.2.9 IBM

A group of people at IBM (including A. L. Brown, G. Heidorn, A. Malhotra, M. Mikelsons, P. B. Sheridan, and I. Wladawski) are working on a system able to explain the programs produced by a customizer [Malhotra & Sheridan 1976; Mikelsons & Wladawski 1976]. The customizer produces programs written in the language BDL [Hammer, Howe, Kruskal & Wladawski 1975]. The system is designed to be able to respond in English to questions posed in English about the programs produced by a customizer.

Their system has a semantic network which contains a model of the relevant business domain. A program is understood by linking it up with the semantic network. The most common type of link indicates the meaning of a data item. For example, that the datum carried by a certain variable is a "price on an invoice." The system does not try to create these links by itself. They are created by the user mostly through the use of mnemonic variable names.

From the point of view of this paper, their system is interesting because it attacks the problem of understanding a program. However, it does not use plans as such. The special nature of BDL allows programs written in it to be directly used to answer many types of questions without reference to a separate plan structure.

II. THE TASKS THE SYSTEM CAN PERFORM

This system can do three things. First, it can develop an understanding of a mathematical FORTRAN program by looking at the program and its annotation. Second, the system can demonstrate its understanding by answering questions about the program. Third, it can use its understanding to aid in the debugging and modification of the program.

In this section, understanding is indirectly defined through the assumption that the system must understand a program in order to perform the second and third tasks. Section III describes how the system acquires and represents an understanding of a program.

Section II.1 defines the second task by specifying in detail the types of questions the system can answer about a program it understands. The remaining subsections describe some ways that the system can utilize its understanding of a program.

The discussions below are concretized through reference to a specific example program (see the figure below). A simpler program is subjected to exhaustive analysis in section III.4.

Fig. 2: The subroutine RK1 from the IBM SSP. All of the examples in section II are taken from this program.

```
1     C     PURPOSE
2     C        INTEGRATES A FIRST ORDER DIFFERENTIAL EQUATION
3     C        DY\DX=FUN(X,Y) UP TO A SPECIFIED FINAL VALUE
4     C     DESCRIPTION OF PARAMETERS
5     C        FUN -USER-SUPPLIED FUNCTION SUBPROGRAM WITH
6     C             ARGUMENTS X,Y WHICH GIVES DY\DX
7     C        HI  -THE STEP SIZE
8     C        XI  -INITIAL VALUE OF X
9     C        YI  -INITIAL VALUE OF Y WHERE YI=F(XI)
10    C        XF  -FINAL VALUE OF X
11    C        YF  -FINAL VALUE OF Y
12    C        ANSX-RESULTANT VALUE OF X
13    C        ANSY-RESULTANT VALUE OF Y
14    C             EITHER ANSX=XF OR ANSY=YF DEPENDING
15    C             ON WHICH IS REACHED FIRST
16    C        IER -ERROR CODE
17    C             IER=0 NO ERROR
18    C             IER=1 STEP SIZE IS ZERO
19    C     REMARKS
20    C        IF XI IS GREATER THAN OR EQUAL TO XF, ANSX=XI AND ANSY=YI
21    C        IF HI IS ZERO, IER=1, ANSX=XI, AND ANSY=0.0
22    C     METHOD
23    C        USES FOURTH ORDER RUNGE-KUTTA INTEGRATION
24    C        PROCESS ON A RECURSIVE BASIS.  PROCESS IS
25    C        TERMINATED AND FINAL VALUE ADJUSTED WHEN
26    C        EITHER XF OF YF IS REACHED
27    C
28          SUBROUTINE RK1(FUN,HI,XI,YI,XF,YF,ANSX,ANSY,IER)
29    C     IF XF IS LESS THAN OR EQUAL TO XI, RETURN (XI,YI)
30          IER=0
31          IF (XF-XI) 11,11,12
32       11 ANSX=XI
33          ANSY=YI
34          RETURN
35    C     TEST INTERVAL VALUE
36       12 H=HI
```

```
      IF (HI) 16,14,20
   14 IER=1
      ANSX=XI
      ANSY=0.0
      RETURN
   16 H=-HI
C     SET XN=INITIAL X, YN=INITIAL Y
   20 XN=XI
      YN=YI
C     INTEGRATE ONE TIME STEP
      HNEW=H
      JUMP=1
      GOTO 170
   25 XN1=XX
      YN1=YY
C     COMPARE XN1 (=X(N+1)) TO X FINAL AND BRANCH ACCORDINGLY
      IF (XN1-XF) 50,30,40
C     XN1=XF, RETURN (XF,YN1) AS ANSWER
   30 ANSX=XF
      ANSY=YN1
      GOTO 160
C     XN1 GREATER THAN X FINAL, SET NEW STEP SIZE AND
C     INTEGRATE ONE STEP, RETURN RESULTS AS ANSWER
   40 HNEW=XF-XN
      JUMP=2
      GOTO 170
   45 ANSX=XX
      ANSY=YY
      GOTO 160
C     XN1 LESS THAN X FINAL, CHECK IF (YN,YN1) SPAN Y FINAL
   50 IF ((YN1-YF)*(YF-YN)) 60,70,110
C     (YN1,YN) DOES NOT SPAN YF, SET (XN,YN)=(XN1,YN1), REPEAT
   60 YN=YN1
      XN=XN1
      GOTO 170
C     EITHER YN OR YN1 =YF, CHECK WHICH AND RETURN PROPER (X,Y)
   70 IF (YN1-YF) 80,100,80
   80 ANSY=YN
      ANSX=XN
      GOTO 160
  100 ANSY=YN1
      ANSX=XN1
      GOTO 160
C     (YN,YN1) SPANS YF. TRY TO FIND X VALUE ASSOCIATED WITH YF
  110 DO 140 I=1,10
C     INTERPOLATE TO FIND NEW TIME STEP AND INTEGRATE ONE STEP
C     TRY TEN INTERPOLATIONS AT MOST
      HNEW=((YF-YN)/(YN1-YN))*(XN1-XN)
      JUMP=3
      GOTO 170
  115 XNEW=XX
      YNEW=YY
C     COMPARE COMPUTED Y VALUE WITH YF AND BRANCH
      IF (YNEW-YF) 120,150,130
```

```
 91     C      ADVANCE, YF IS BETWEEN YNEW AND YN1
 92       120 YN=YNEW
 93           XN=XNEW
 94           GOTO 140
 95     C      ADVANCE, YF IS BETWEEN YN AND YNEW
 96       130 YN1=YNEW
 97           XN1=XNEW
 98       140 CONTINUE
 99     C      RETURN (XNEW,YF) AS ANSWER
100       150 ANSX=XNEW
101           ANSY=YF
102       160 RETURN
103     C
104       170 H2=HNEW/2.0
105           T1=HNEW*FUN(XN,YN)
106           T2=HNEW*FUN(XN+H2,YN+T1/2.0)
107           T3=HNEW*FUN(XN+H2,YN+T2/2.0)
108           T4=HNEW*FUN(XN+HNEW,YN+T3)
109           YY=YN+(T1+2.0*T2+2.0*T3+T4)/6.0
110           XX=XN+HNEW
111           GOTO (25,45,115),JUMP
112           END
```

II.1 ANSWERING QUESTIONS ABOUT A PROGRAM

This section describes, mainly through examples, several classes of questions which cover most of the questions one might ask about a program. The system is designed to be able to answer all these types of questions.

The examples of what a user might ask (italics) about the program RK1, and of what the system might respond are given in English prose. This is not intended to indicate that the system will be able to converse in English. That is a separate problem. The system will communicate in some, as yet unspecified, LISPese dialect. English is used in this proposal for clarity of exposition.

Another point which should be raised, is that the questions and answers often refer to segments of code (denoted by the line numbers of all the lines contained in the segment). It is reasonable to ask what constitutes a meaningful segment. The answer to this question is a function of the way the system understands programs. It is given, in detail, in section III (particularly III.1.1 and III.4). Basically, a reasonable segment is a section of the program that can be looked at as being a program in its own right. It has inputs and outputs. Further, control enters at one point, and leaves for one point.

In the examples, only reasonable segments are mentioned. If a user asked about an unreasonable segment, the system would tell him that the segment was not reasonable and try to give him a better idea of the segmentation so that he could ask a better question.

II.1.1 REQUESTS FOR DESCRIPTION, "WHAT"

The most basic kind of question which can be asked is, "What is this?" The question does not ask how the thing works or what its place is in the grand scheme. The question just asks for a description of its behavior. Consider some examples.

1) *What is line 30 of the program RK1?*
   It is an assignment statement forming part of a data flow path to communicate 0 to the outside of RK1 via the variable IER.

2) *What is line 31?*
It is an ARITHMETIC IF statement forming part of the control flow connective tissue.

3) *What is "XF-XN" in line 60?*
It is a use of the primitive function minus which computes a=b-c.

Questions such as the above are not very interesting. This is because it is obvious to us what such small pieces of the code do. It should be noted, however, that it is important that the answers to these questions are also obvious to the system. Even if no user ever asks such questions, the system will ask itself such questions when it is trying to answer more difficult questions.

"What" questions become more interesting when they are asked about larger sections of the code or from a non-local point of view.

4) *What does this segment (lines: 104-110) do?*
It performs one step of integration.

5) *In more detail, please.*
Starting with XN, YN, HNEW, and FUN it computes XX=XN+HNEW and YY=F(XX) given that FUN(X,F(X))=dF/dx(X), and YN=F(XN).

Note that, in this answer, the system used the variable names in the program as names for the data items it was talking about. Internally is does not name them that way. It realizes that the variables are only part of the data flow connective tissue, and are not satisfactory as names for the data items because one variable often carries logically unrelated data items in different parts of the program. This notwithstanding, it is probably better to use the variable names when talking to the user than ad hoc new names for the data items.

6) *What does this segment (lines: 60-64) do?*
It computes ANSX=XF and ANSY=F(ANSX) starting with XN, YN, and FUN given that FUN(X,F(X))=dF/dx(X), and YN=F(XN).

7) *From the point of view of that segment (lines: 60-64) what does "XF-XN" in line 60 do?*
It computes HNEW such that XN+HNEW=XF.

"What" questions are answered through reference to the intrinsic and extrinsic behavioral descriptions of segments (behavioral descriptions are discussed fully in section III.2.1). The difference between intrinsic and extrinsic behavioral descriptions is illustrated by the difference between the answers to questions 3 and 7.

When asked about large segments, the answers to "what" questions are not obvious. The answers to questions like 4-7 could aid a user in understanding the program RK1. Still, the answers to these questions do not explain the program, they only describe it.

II.1.2 REQUESTS FOR EXPLANATION, "HOW"

"How" questions ask for an explanation of the internal workings of a piece of a program. For example, "How does this work?" A related set of questions asks how a piece of a program interacts with larger pieces containing it. For example, "When is this executed?" and "Where do its inputs come from?" These questions are really just asking for an explanation of an aspect of the workings of the containing segment. Consider some examples.

8) *How does line 30 work?*
Assignment is a primitive operation which copies a value (here 0) to a variable (here IER). It forms a link in a data flow path.

9) *When is line 30 executed?*
It is executed once every time RK1 is executed. However, logically it should only be executed when XI≥XF ∨ HI≠0. The line was factored to its current position from that segment (lines: 32-33) and that segment (lines: 36-37,42-101). When XI<XF ∧ HI=0 the effects of line 30 are overridden by line 38.

Note that the system volunteered information about the factoring of line 30. It did this because it understands that the simple statement "Line 30 is executed once each time RK1 is." is misleading.

10) *How does line 31 work?*
It uses the FORTRAN control flow construct ARITHMETIC IF to convert the segment "XF-XI" into a predicate determining whether XI<XF or not.

11) *How does this segment (lines: 60-64) work?*
The basic segment (lines: 60-62) is a composition of line 60 (which determines HNEW so that XF=XN+HNEW) and that segment (lines: 104-110) (which performs a step of integration yielding XX=XN+HNEW=XF and YY=F(XX)). Lines (61,62,111) are control flow connective tissue implementing the link to that segment (lines: 104-110). Lines (63,64) are data flow connective tissue passing the results along.

12) *When is this segment (lines: 60-64) used?*
When that loop (lines: 44-53, 67-71) terminates with XN1>XF, this segment (lines: 60-64) is used to compute the results of the program RK1.

These questions are answered through reference to plans (see section III.2.2). The answers describe how segments interact to form larger segments. The next two sections describe questions which get at specific aspects of this interaction.

II.1.3 REQUESTS FOR PURPOSE, "WHY"

A question of this type asks for the purpose of a construct. It asks why it is in the program.

13) *What is the purpose of IER in line 30?*
The purpose of this use of the variable IER is to form part of a data flow path carrying the value 0 to the outside of RK1.

14) *What is the purpose of this data flow path (the one implemented by line 30)?* In the situations where XI≥XF ∨ HI=0, this path carries one of the outputs (0) of the program, as required by lines (17, 18, 20, 21). It is incidental (not part of its purpose), that it also starts to carry 0 when XI<XF ∧ HI=0. The value is overridden in this case.

15) *Why is line 31 in the program?*
This predicate determines whether XI<XF or not. This is done so that the computations requested of the program RK1 can be divided into two classes, one where XI<XF and one where XI≥XF.

16) *Why is this segment (lines: 60-64) in the program?*
The purpose of this segment is to compute ANSX and ANSY, two of the results of the program RK1, in the situation where that loop (lines: 44-53, 67-71) terminates with XN1>XF.

Note the similarity between the answer to this question and the answer to question 12. The purpose of a segment and the plan for the containing segment are heavily intertwined. Asking "when" is often equivalent to asking "why".

17) *What is the purpose of line 37?*
Due to the fact that several tests have been combined into this one construct, it serves three distinct purposes. First, it determines whether HI=0 or not. This is used to divide the computations requested of the program RK1 into two classes. Second and third, in that segment (lines: 36, 37, 42), this line (37) implements two predicates. One checks whether HI<0 and the other checks whether HI>0. These are used to divide the problems faced by that segment (lines: 36, 37, 42), which computes the absolute value of HI, into two classes.

Here it is seen that one piece of code may have several distinct purposes.

18) *Why is line 60 in the program?*
It is composed with that segment (lines: 104-110). Its output HNEW is computed so that when HNEW is input to that segment (lines: 104-110) that segment will produce an output XX=XF.

These questions are answered through reference to purpose links in plans (see section III.2.2).

II.1.4 REQUESTS FOR JUSTIFICATION, "WHY"

These questions ask for the reason why something is true. Often something is true because it is some segment's purpose to make it true. In this case, a justification question is just a purpose question from a different point of view. However, only the main goals of a segment are recorded as purposes. A feature of a segment may be used to justify something even though the feature is not considered as satisfying a purpose.

19) *Why is the prerequisite of line 60 that XF be a floating number satisfied?*
It is satisfied because the same requirement is a prerequisite of the immediately containing segment (lines: 60-64) and its satisfaction carries over.

20) *Why is this prerequisite (of lines: 60-69) satisfied?*
Its satisfaction is guaranteed by a chain of prerequisites reaching back to the prerequisites of RK1. The value of XF is carried unchanged from an argument to the program RK1 which is required to be a floating number.

Seeing that the user is interested in a full justification for the prerequisite, the system traces back as far as possible in order to give a definitive answer. There is no better justification than the above. If RK1 is called with the argument XF which is not a floating number, it will not work.

21) *Consider the use (lines: 61, 62) of that segment (lines: 104-110) in this segment (lines: 60-64). Why is the prerequisite that HNEW be a floating number satisfied?*
It is satisfied because an output assertion of line 60, the source of HNEW, states that HNEW is a floating number.

22) *What justifies the output assertion of this segment (lines: 60-64) that ANSY=F(ANSX)?*
Data flow (lines: 63-64) makes ANSX=XX and ANSY=YY. It is an output assertion of that segment (lines: 104-110), the source of XX and YY, that YY=F(XX). Therefore, ANSY=F(ANSX).

23) *What justifies the output assertion of this segment (lines: 60-64) that ANSX=XF?*
Data flow (line: 63) makes ANSX=XX. It is an assertion of that segment (lines: 104-110), the source of XX, that XX=XN+HNEW. It is an assertion of line 60, the source of HNEW, that XN+HNEW=XF. Therefore, ANSX=XF.

Note that this is about as difficult a proof as this system can make. It tries to prove things correct by pattern matching, and false by testing them on examples. If a more complex proof is required, when the system is developing its understanding of a program, the system just believes the implicit claim of the program writer that the proof is possible. For example:

24) *What is the justification of the output assertion of this segment (lines: 104-110) that YY=F(XX)?*
The prerequisites of this segment (lines: 104-110) guarantee that FUN(X,F(X))=dF/dx(X), and YN=F(XN). The program writer claimed that it was a true theorem that if these prerequisites were met then evaluating the equation implemented in lines (104-110) would yield YY=F(XX).

Note that it is actually not a true theorem. The equations only approximate the integral over the interval from XN to XN+HNEW if HNEW is sufficiently small. Even this is not easy to prove. When trying to understand the program RK1 it is helpful to assume that the equations actually calculate the integral, though this would be impossible to prove since it is in fact false. The approach taken here avoids getting involved in the mathematical complexities of approximation.

These questions are answered through reference to reason links in plans (see section III.2.2).

It is felt that the ability to answer the types of questions illustrated in the last four sections would show that the system can understand a program. The next sections indicate some tasks the system could perform beyond explaining a program.

II.2 AIDING THE DEBUGGING PROCESS

Any difference between the operation of a program, and the operation intended by the programmer is a bug. This system is designed to aid a programmer in detecting and eliminating any differences.

In order to detect differences between the operation of a program, and the operation intended by the programmer, the system must understand both of them. Section III.4 describes how this is done.

To determine the operation of the program, the system starts with direct analysis of the code based on its knowledge of the primitive constructs. To determine the operation intended by the programmer, the system starts with direct analysis of the programmer's annotation of the program, and with assumptions about what modes of operation can possibly be intended (for instance, that in the SSP, no programmer ever intends to divide by zero).

Both of these approaches bog down short of complete understanding. The annotation, though fairly easy to understand, is always incomplete. The code, though always completely specifying the operation of the program, is too complex to be understood without guidance. Full understanding is developed by assuming that the two descriptions do describe the same operation. Each is used as a guide to fill in the gaps in the other.

The result of this process is a single entity, the grand plan. It specifies what the system thinks

the operation intended by the programmer is, and exactly how the program implements this operation.

The system cannot represent both the operation of the program and the intended operation of the program with one description if it thinks there are any differences between them. Any differences it ever finds are reported as bugs.

II.2.1 FINDING BUGS STATICALLY

The system uses the programmer's annotation as a guide for developing a description of the operation of a program. After this is accomplished, the annotation, and general knowledge about what modes of operation are reasonable, are checked against the operation of the program. The system tries to justify that the intentions of the programmer are realized in the program.

Looking at a particular claim, the system may prove that it is valid. In that case, fine. On many occasions, the system will not be able to come to any conclusion about the validity of a claim. In that case, the system just believes the claim is true, while remembering that it is unsubstantiated.

Alternately, the system may be able to prove that the claim is invalid. In that case, the system reports this as a bug. These bugs, which are found via pre-execution analysis of the program, are the subject of this section.

There are several general claims which can be assumed to be intentions of the programmer (at least with regard to simple mathematical FORTRAN programs). These claims can be used to uncover many bugs.

One claim is that the extrinsic prerequisites of every segment must be satisfied. This is to say that no programmer will deliberately use a segment with inputs outside its stated domain of applicability. Testing this claim at the beginning of every use of every segment leads to the detection of bugs such as incompatible subroutine arguments, using the wrong variable name, leaving out special case checks, etc.

Another general claim is that the satisfaction of the extrinsic prerequisites for the use of a segment must imply the satisfaction of the intrinsic prerequisites of the segment. In a similar vein, the intrinsic assertions of a segment must imply the extrinsic assertions of that segment. That is to say, the segment must be capable of what it has been asked to do. These claims uncover bugs based on a misunderstanding of the inherent abilities of a segment.

In addition, there are other more specific claims. Claims such as loops must terminate; uninitialized variables cannot be read or returned, etc. Many of these can be handled by properly stating the prerequisites of the primitive constructs.

The really interesting bug detection involves comparing what the programmer said should happen with what does happen. Consider the example program, RK1. Inasmuch as it is a published program that people have been using for years, it has no bugs in it which can lead to catastrophic failure of the program. However, it does not operate in the manner that the comments clearly indicate that it should.

For example, an examination of the program readily shows that the comment in lines (20-21) does not correspond with the operation of the program. The comment clearly states that XI≥XF → ANSY=YI and that HI=0 → ANSY=0.0. The way the program operates is that XI≥XF → ANSX=YI and that ANSY=0.0 only if HI=0 ∧ XI<XF. It can be seen that this is a bug in the comment, not in the program. The mode of operation indicated by the comment is impossible since XI≥XF and HI=0 are not mutually exclusive conditions. It is also clear that this is a minor bug, as long as some other programmer does not take the comment seriously and write another program which depends on the fact that HI=0 → ANSY=0.0.

A more serious bug is involved with the search scheme embodied in lines (81-98). This segment uses repeated interpolations to seek out XNEW such that F(XNEW)=YF. It is basically just a slightly improved version of a halving search (in which line 84 would be HNEW=(XN1-XN)/2). The key to the method is that the interval (XN,XN1) always contains a root of F(X)-YF and that the size of the interval decreases at each step. The simple halving search is guaranteed to improve the accuracy of the result by a factor of 1024 in 10 iterations.

The search starts with YF in the interval (YN,YN1) (note the comment line 80). A new value of Y is calculated (YNEW) and then the test in line 90 is used to determine whether YF is in the interval

(YN,YNEW) or the interval (YNEW,YN1). This intention is indicated by the comments in lines (89,91,95). The problem is that the particular predicate chosen to make the determination will work correctly only when YN<YN1. If YN>YN1, it consistently makes the wrong choice. As a result the search will only work in the intended fashion when YN<YN1. This, however, is not assured.

The form of the test in line 67, which performs a similar function, indicates that the programmer did not intend to limit RK1 to working on monotonically increasing functions only. Therefore it is probably the test in line 90 which is in error, and should be changed.

With this bug, the interval (XN,XN1) is no longer guaranteed to decrease in size, or to even contain a root of F(X)-YF. This would lead to considerable trouble if it were not for the fact that the search segment is robust. First, line 84 can extrapolate as well as interpolate. Second, the segment implemented by lines (104-110) will work fine with HNEW<0. Third, the segment implemented by lines (90-97) tends to create an interval (YN,YN1) containing a root of F(X)-YF where YN<YN1. As a result of this, after thrashing in the first couple of iterations, the search begins to work more or less correctly. This is probably why this bug was never found. Only rare pathological functions can cause catastrophic failure of the search.

This is an example of a definite bug that can be detected by comparing the operation of the program with the intentions of the programmer, but which is almost impossible to detect by looking at the performance of the program RK1 on test data.

The program RK1 has several other bugs similar to the two cited above.

II.2.2 FINDING BUGS DYNAMICALLY

The system reports a bug statically when it is able to disprove a general claim, or a claim made by the programmer, about the program. However, the system does not have an elaborate deduction mechanism. It uses mainly pattern matching and trial by example. As a result, it is often reduced to accepting a claim on faith.

The proposed system has the added ability to execute a program in a careful mode where it continually checks all the claims it was not able to prove valid. This is essentially just letting a user of the program indirectly suggest what data items should be used in order to check the claims by means of trial by example.

If a discrepancy is discovered, the system proceeds basically the same way as if the bug had been found statically through a fortuitous choice of trial by example. The only difference is that the partial computation can be used to help understand the bug.

Suppose that the two bugs described in the last section were not found statically. The first bug would be detected dynamically the first time RK1 was called with HI=0 and XI<XF. Similarly, the second bug would be found the first time that the search was initiated with YN1<YN.

II.2.3 UNDERSTANDING BUGS

The last two sections talked about detecting the existence of a problem. Once a problem is detected it must be traced back to its source so that it can be corrected.

Most computing environments do a certain amount of dynamic bug detection. For instance, they continually check claims such as: non-existent memory will not be referenced, and division by zero will not be attempted. If, for example, an illegal memory reference occurs, the computing environment reports this as a problem. The programmer then begins the involved process of finding the source of the bug. This system starts out more than half way done with this process because the problems it finds are at a much higher logical level.

A useful system should be able to converse with a user about a bug, just as it can about a program. In addition, it should be able to present the issues involved so that the user can make decisions about what should be changed.

The system determines what the relevant issues are by asking itself questions. It asks whether there was any justification for the claim which has just turned out to be false. If so, this justification is obviously spurious. The system tries to see whether any part of it has an obvious weakness. Depending on the type of error, the system determines what the other segments involved are. For

instance, what is the source of the variant input or who will receive the variant output.

Consider the first bug in RK1 discussed above. Suppose that the system discovers statically that when XI≥XF and HI=0, ANSY=YI though the comment in line 21 claims that ANSY should equal 0.0. The system might report this as follows:

> There is a descrepancy between the value of ANSY returned by the program, and the comment on line 21. When XI≥XF and HI=0, ANSY=YI which is not neccessarily equal to 0.0 even though HI=0.

The above identifies the error, but it is not enough. The system should present a more complete description of what is going on as follows:

> The top level structure of the program RK1 is a COND XOR (see section III.3.2.2).
> "[IF XI≥XF THEN ... ELSE [IF HI=0 THEN ... ELSE ...]]"
> This implies that (HI=0 ∧ XI<XF) → ANSY=0.0.
> On the other hand, the comments on lines (20,21) indicate that the top level structure of the program was intended to be either an AND (see section III.3.1.1).
> "[IF XI≥XF THEN ...] AND [IF HI=0 THEN ...] AND ... "
> Or a CASE XOR (see section III.3.2.1).
> "select the one applicable case and perform the corresponding computation
> CASE1 XI≥XF THEN ... CASE2 HI=0 THEN ... CASE3 XI<XF ∧ HI≠0 THEN ... "
> With either structure HI=0 should always imply that ANSY=0.0.

This describes the true nature of the descrepancy between the program and its annotation. The system should go on to volunteer added relevant information such as:

> XI≥XF and HI=0 are not mutually exclusive situations.
> As a result, a CASE XOR is not reasonable as the top level plan for RK1.
> Further, when both conditions hold, it is not possible for ANSY to equal 0.0 as required by line 21 and to also equal YI as required by line 20 when YI≠0.0, no matter how the program is implemented.

This last statement points directly at the cause of the problem. The system has noticed that the comments on lines (20,21) are actually contradictory. The most reasonable way to fix this bug is to change the comments since they are unreasonable.

The system should give a similar in depth analysis of the second bug in RK1. Suppose that the system found this bug dynamically, it might say something such as:

> The comment on line 91 indicates that it is a prerequisite of the segment implemented by lines (92,93) that YF be between YNEW and YN1. In the current situation this is not the case even though execution of the segment implemented by lines (92,93) is about to begin.
> It should be noted that the test on line 90 tested only that YNEW<YF. The programmer implied that it could be proven that YF is between YNEW and YN1 whenever YF was between YN and YN1 and YNEW<YF. This is obviously not the case.

The system first describes the conflict which has arisen. Then it identifies the theorem which had been used to justify that XF whould indeed be between YNEW and YN1.

It should be noted that the system probably would not be able to develop such a pithy justification from the program as it stands. The system would start with a justification based on all of the prerequisites of the segment (lines: 81-98), all of the assertions of the segment (lines: 84-88), and the assertions of line 90. This contains a lot of irrelevant information. In order to distill this down

to a concise justification, the system would probably need assitance from the user. For instance, the system might ask, "Why does the fact that YNEW<YF from line 90 imply that YF is between YNEW and YN1, as required by line 91?" The user might answer "Because YF is between YN and YN1."

Another piece of information the system could give the user would be a counter example. The system could use the values that the variables currently have from the partial computation to guide it in the selection of a counter example to the above theorem which would illustrate the problem.

> For example, if YN=4, YF=3, YN1=1, and YNEW=2 then clearly YF=3 is between YN=4 and YN1=1 and YNEW=2 is less than YF=3, however, YF=3 is not between YNEW=2 and YN1=1.

The user could ask for more information if he desired, in order to decide whether the test on line 90 should be changed, or a prerequisite requiring that YN<YN1 should be added to the search.

II.3 UNDERSTANDING MODIFICATIONS

There are two main aspects of modification: deletion and insertion. When something is deleted, the system must ask itself what depended on the existence of the deleted section, what loose ends have been left dangling. If there are no loose ends, then the section must not have been used for anything.

When a section is inserted the system must ask itself two questions, "What contribution is this section going to make?" and "What problems is it going to cause?" The system can get hints about the answer to the first question by knowing that a section is replacing another, or supposedly fixing a bug.

There are many potential problems that a modification may cause. Control or data flow paths may be disrupted. Needed assertions may be deleted or contradicted. This could cause justifications to become invalid. It can be seen that a lot of work is required in order to assess the ramifications of a modification. However, it should also be noted that a lot of this same type of work is required when the system analyzes the program in the first place.

Consider some examples of how the system might respond to a suggested modification.

> *25) What would happen if line 30 were deleted?*
> In the situations where XI≥XF or HI≠0, the argument to RK1 corresponding to the variable IER would be returned unmodified. This contradicts the comments on lines (17,18,20,21) which states that IER should be 0.
>
> *26) What would happen if line 31 were changed to "IF (XF-XI) 11,12,12"?*
> The new program would be different in that this segment (lines: 32-33) would only be executed when XI>XF and the rest of the program would be executed when XI=XF as well as when XI<XF.
> When XI=XF and HI=0, then IER would equal 1 and ANSY would equal 0.0 whereas, IER=0 and ANSY=YI in the unmodified program.
> When XI=XF and HI≠0, it seems (though I (the system) cannot prove it) that it is still the case that ANSX=XI and ANSY=YI as required by the comment on line 20.
>
> *27) What is your justification for the last statement?*
> Assuming XI=XF and HI≠0, then for all of the examples I (the system) have tried, that loop (lines: 44-53, 67-71) terminates on the first iteration with XN1(=XI+HNEW)>XF. This causes this segment (lines: 60-64) to be executed. Again for all of the values I (the system) have chosen, HNEW=0 and therefore ANSX=XX=XN+0=XI=XF and ANSY=YY=F(XX)=F(XI)=YI.

It is entirely possible that the system would actually be able to prove the assertion. It is

interesting, however, that the very simple deductive methods, trial by example and substitution of equals, lead to a such lucid explanation of what is happening.

Now consider modifications which might be proposed as fixes for the two bugs in RK1 discussed above.

28) What would happen if line 21 were changed to
"C OTHERWISE IF HI=ZERO THEN IER=1, ANSX=XI, AND ANSY=0.0"?
This would eliminate the bug by removing the conflict between the old line 21 and the top level structure of the program.

Note, comments as well as code may be modified.

29) What if the test in line 90 were changed to
"IF (YF-YNEW) 120,150,130"?
I (the system) assume you intend to claim that it is true that when XF is between YN and YN1 and YF<YNEW then YF is between YNEW and YN1.
This is true for the example which brought up the bug. However, it also has counter-examples for instance, when YN=1, YF=2, YNEW=3 and YN1=4.

30) What if the test line 90 is changed to
"IF ((YNEW-YF)(YF-YN1)) 130,150,120"?*
As far as I (the system) can tell, this fixes the bug without disrupting any of the other uses of line 90.

III. HOW THE SYSTEM WORKS

This paper is a proposal. As a result, this section claims to be neither complete nor completely accurate. It only tries to show that it should be possible to achieve the behavior described in the rest of this paper. With this in mind, consider the following description of how the system might work.

The system constructs a logical structure, the grand plan, for a program. This structure shows how the problem of achieving the goal of the entire program is reduced, through the program, to subproblems which can be achieved by primitive programs available in the programming environment.

The grand plan is a tree. Each node is a plan explaining how a goal is decomposed into subgoals, the plans for which are the daughters of the node. The leaves are goals achievable by primitive programs.

Each part of the program is linked with the piece(s) of the grand plan, which it implements. Each part is explained by this link. Questions about a part of the program are referred to the corresponding part of the grand plan in order to be answered.

III.1 DECOMPOSITION OF A PROGRAM

A program is decomposed into sections following the structure of its grand plan.

III.1.1 SEGMENTS

The goal associated with a node, or leaf, of the grand plan is referred to as a "segment of the main goal." The set of parts of the program which are associated with this node, and its decendants, is referred to as a "segment of the program," the program segment which implements the goal segment. In this paper the term "segment" is used to refer to both the goal segment and the corresponding program segment.

In general, a program will have a basic tree-like structure closely parallel to the structure of its grand plan. However, many transformations are commonly applied to a program in order to increase its efficiency (see section III.1.3). These transformations obscure the basic parallel. As a result, a program segment need not be a simple continuous piece of the program. It may be spread here and there through the code. In addition, one piece of code may be contained in several logically non-overlapping segments.

The code for a given segment, with the rest of the program deleted, forms a new program which implements the corresponding goal. Further, it is a property of the way this system chooses segments that each segment has only one entry and only one exit path. In other words, no information is encoded in the flow of control into or out of a segment. Each segment of a program is described by a behavioral description (see section III.2.1).

III.1.1.1 A SMALL EXAMPLE

This section presents an example of the relationship between a program and its grand plan. Section III.4.8 discusses tha grand plan for a more complex program.

Consider a segment of code which computes the roots of a quadratic polynomial which is assumed to have real roots. It takes as input the coefficients A, B, and C and outputs the roots R1 and R2.

```
D  = SQRT(B**2-4*A*C)
R1 = (-B+D)/(2*A)
R2 = (-B-D)/(2*A)
```

Fig 3: A program which finds the roots of a quadratic polynomial.

The following is a possible grand plan for this program. Each node has a name of the form Gn.

The node specifies the goal which it achieves, and a plan for how to achieve it.

```
G1: "calculate the roots" to do this achieve G2 and G3
        G2: "calculate first root" compose the goals G4 and G5
                G4: "calculate the square root term" D=SQRT(B**2-4*A*C)
                G5: "get the first root" R1=(-B+D)/(2*A)
        G3: "calculate second root" compose the goals G6 and G7
                G6: "calculate the square root term" D=SQRT(B**2-4*A*C)
                G7: "get the second root" R1=(-B-D)/(2*A)
```

Fig 4: A grand plan for finding the roots of a quadratic polynomial

The final figure in this section shows how the program is segmented, and how these segments correspond to goals in the grand plan.

```
     program segments          code
P1 P2 P3 P4    P6       D  = SQRT(B**2-4*A*C)
P1 P2       P5          R1 = (-B+D)/(2*A)
P1    P3          P7    R2 = (-B-D)/(2*A)
```

Fig 5: This shows the correspondence between the program and the grand plan. The program segment Pi achieves the goal Gi. The program segment name appears before each line contained in it.

Note that due to factoring (see section III.1.3.3), segment P3 is not continuous, and the first line implements both P4 and P6, which are logically non-overlapping segments.

III.1.2 CONNECTIVE TISSUE

Consider a program segment P corresponding to a goal G. The segment P contains subsegments Pi which achieve the subgoals Gi of G. The segment P also contains code which is not contained in any subsegment. This excess code is connective tissue. It is the cement which binds subsegments together to form a larger segment achieving a more complex goal.

The goal of a segment is achieved by executing a set of subsegments. However, the subsegments are not just executed in a vacuum. They must have information conveyed to them, from them, and between them. In addition, they must be executed in the correct sequence.

III.1.2.1 DATA FLOW CONNECTIVE TISSUE

Data flow connective tissue carries data items between segments. It carries data from the output of one segment to the intput of another segment, from the output of a subsegment to the output of the containing segment, and from the input of a segment to the input of a subsegment.

The most common data flow constructs which form these pathways are subroutine arguments, returned values, free and bound variables, and assignment. These constructs can be chained together so that a datum can follow a path consisting of many sections from its source to its destination.

III.1.2.2 CONTROL FLOW CONNECTIVE TISSUE

Static control flow connectives, such as GOTO, CALL, RETURN, and physical sequential placement of statements, fix a pattern of execution. In addition, dynamic control flow connectives, such as DO and various IFs, perform computations by varying the pattern of execution. In this system, the key actions of dynamic control flow are captured in the notion of a predicate, which is used explicitly in several plan types.

A predicate is a hybrid construct formed by wrapping a segment in dynamic control flow

constructs. A predicate has no explicit outputs, however it causes control to exit via one of two paths. On one path it makes the output assertion P and on the other ¬P. The output is in effect encoded in the flow of control.

Predicates are only used in certain plan types, where the way they contribute to the overall goal is made explicit. This makes it easier to understand the function of predicates.

III.1.3 PROGRAM TRANSFORMATIONS

One of the greatest problems in recognizing the basic structure of a program is that, for a variety of reasons, pieces of the code are shuffled around when the basic plan is implemented. The primary motivation for this is economy. Pieces are moved so that they need not be redundantly executed. Pieces are also often moved so that they need not be redundantly written, even if this increases execution time.

The transformed program performs the same essential calculation. However, it usually has incidental differences, complexities, and redundancies which may lead to bugs when it is modified, or incorporated in a larger program.

This system attempts to detect transformations as it develops the grand plan for a program. It continually tries to get at the underlying logical structure of a program.

III.1.3.1 REARRANGEMENT

Consider the primitive pieces of code which form a program. They form indivisible units implementing the logical segments which are leaves of the grand plan. Connective tissue links these pieces together into a program. The control flow connective tissue forms a directed graph, each node of which is executed only in a certain set of situations.

A given piece of code can usually be put in any of several positions in the flow of control. In fact, the placement is constrained in only two ways. First the data links provide a partial ordering for the fundamental pieces of code. A piece of code must be situated so that it follows (in the order of execution) any piece of code which provides data for it, and precedes (in the order of execution) any piece of code which uses data output from it. Secondly, the grand plan specifies what subset of situations each piece of code should be executed in. A piece of code is constrained to positions in the control flow which cause execution only in the correct situations. That is to say, a piece of code which should only be executed some of the times that the program as a whole is executed (for example one of the alternatives of an IF), must be put in a section of the topological control flow structure which is executed at just those times.

Clearly the grand plan only loosely restricts the position of the pieces of code. One reasonable way to decide on the exact sequence is to require that pieces of code which implement the same logical segment be together. This makes the relationship between the grand plan and the program clear, but it is sometimes wasteful.

It should be mentioned that programmers often desire even more flexibility in placement than that which is described above. There is no way to get around the data flow constraints except for changing the algorithm, and hence the grand plan. However, the control constraints can be loosened. It is possible to position a piece of code so that it is executed too often, as long as its results can be ignored in the extra cases. For instance, they can be ignored if, in the extra cases, another section of code overrides them.

It should be noted that stretching the grounds for rearrangement in this way is dangerous. It can lead to bugs because it creates extraneous situations which are peripheral to the main goal. Things come into being whose sole function is to fix things up after something has been moved. These things can easily get mislaid or misunderstood, causing bugs.

III.1.3.2 SUBSTITUTION

A section of code can always be substituted for another section of code as long as it does exactly the same thing in the given situation. This is not terribly useful. However, a section of code

can also be substituted if it does more than is required as long as the extra work can be ignored. It can be ignored by either throwing it away, or by overriding it at a later time. This type of substitution is useful in utilizing pre-existing sections of code. It is also useful in promoting factorization (see below).

Just as in rearrangement, stretching the situation can lead to later troubles. For instance, the requirements for ignoring the extra work can easily get lost in the shuffle because they are peripheral to the main task. Also some other program may eventually come to depend on one of the extraneous features. This can lead to future problems, since the original program does not guarantee the extraneous features. They may go away at any time.

There is a special case of substitution which deserves note. Often the grand plan will call for a general routine, however, the situation makes it clear that the actions of the routine will be limited to a subset of its normal actions. In this situation, a more restricted routine can be substituted. Sometimes the routine can be eliminated entirely because its inputs are so heavily constrained that only one of the outputs is possible.

III.1.3.3 FACTORING IN SPACE

Here, two or more identical sections of code are replaced by one instance of the section of code. This saves redundancy in writing the program. Consider two sections of code which have the same specifications, and which are in parallel positions in the flow of control. That is to say, in a given situation either one, but not both, of the sections is executed. The two sections can be replaced by one section (identical to them) in a position that is executed in just the union of the situations that the original two were executed in. In addition, of course, this new position must be such that the data flow restrictions are met.

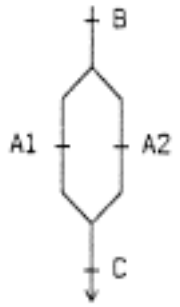


Fig. 6: Two identical pieces of code at A1 and A2 can be factored forward to position B, or backward to position C, as long as the new position is consistent with data flow constraints.

Both of the methods of stretching discussed above can be used here to promote factoring. First, substitution can be used to make dissimilar pieces of code identical. Second, a segment can be factored to a point that is executed too often as long as the results can be ignored in the additional situations. Finally, factoring can be generalized to factoring out of n parallel positions, rather than just two.

It should be noted that factoring is a process which causes one piece of code to perform two or more functions which are logically unrelated. A factored piece of code performs all of the actions which were performed by its antecedents as a group. Since the antecedents were in parallel positions, their actions were associated with logically distinct situations.

III.1.3.4 FACTORING IN TIME

Here a section of code can be moved out of a position, where it is executed many times, to one where it is executed only once, as long as the result of its execution is the same each time it is

executed. The archetypical case of this is moving something out of a loop. However, it should be noted that this type of factoring can also be used when the repetitive serial execution of identical pieces of code, with the same input, is explicit.

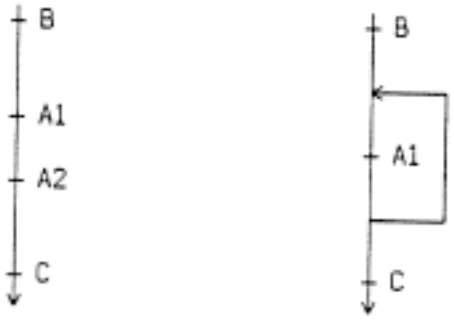


Fig 7: Pieces of code with identical results (such as those at A1 and A2) can be temporaly factored forward to B or backward to C as long as data flow constraints are met.

This process is obviously subject to the same restrictions as spatial factoring. The movements must not violate data flow constraints, and the new position must be executed every time any of the old positions were executed. In addition, as above, things can be stretched and generalized to n pieces of code.

It should be noted that this type of factoring usually only occurs forward. In the figure, if the results of A1 and A2 are not used until after point C, then A1 is clearly redundant.

III.1.3.5 MOVING COMPUTATION BETWEEN THE CODE AND THE FLOW OF CONTROL

There are many things which can either be computed by functions, or directly implemented in the flow of control. A straightforward example of this is a logical connective such as AND or OR. For example, a test for $x<0 \vee y<0$ could be implemented using one logical expression, or by using two separate tests branching to the same place. This system should be able to recognize that these two implementations perform essentially the same computation.

Looking at it another way, position in the flow of control can be used to encode information (for example the outcome of the first test above). A programmer can choose to encode this information in a variable or returned value instead. This usually leads to a simplification of the flow of control. Another good example of this is in the use of loops (see section III.3.4).

If something is to be implemented in the flow of control, there is usually considerable flexibility in the way it can be done. Loops can be implemented with DOs or IFs. Subroutine calls can be implemented with CALLs or computed GOTOs.

In addition, FORTRAN has a triple branching IF. This can often be used to implement two cascaded binary choices. This is particularly confusing when the two choices are logically unrelated.

III.1.3.6 MOVING COMPUTATION BETWEEN THE CODE AND THE DATA FLOW

The basic use of a data flow path is to transmit a datum from one place to another. When this is being done, the datum is not modified by its journey. However, data flow paths can be constructed so that they perform a computation. When this is the case, the datum which exits from the path may not be the same as the one which entered the path.

For example, various devices, such as calling a subroutine with arguments of nonmatching types, can be used to bring different sets of access functions into play at the two ends of a path. If different access functions are used to store and retrieve a datum, it may be transformed by the data path. This system treats this as if explicit transformations had been used. This has the effect of factoring the computation back out of the data flow.

Data flow paths can also be used as a syntactic method for specifying that certain conversions should be performed. For instance the statement "S=D" (where S is single precision and D is double precision) will cause the FORTRAN compiler to insert a conversion from double to single precision. This system just treats this as if the conversion was explicitly stated in the first place.

FORTRAN has the powerful constructs "EQUIVALENCE" and "COMMON" which cause two or more variables to refer to the same section of memory. They can be used to cause different sets of access functions to be used on the same datum, and for other useful things. However, they can also bring about opaque and complex side effects. Initially, this system will not try to deal with the difficulties associated with EQUIVALENCE and COMMON. It will assume that every variable is distinct, i.e. that the value held by a variable can never be modified by an operation on a variable with a different name.

Another issue is brought up by arrays, which are the only non-atomic data item available in FORTRAN. One way to look at an array reference "A(I)" is that it is a function which takes two arguments, the array and the index, and returns a reference to the element of the array. Another way to look at it is as a name for the selected item, and hence a data flow path for that item alone. The second approach has the advantage that, when retrieving and storing a value in an array element, it captures the notion that the other elements of the array are unaffected. In the first view, the array indexing is part of the computation. In the second view, it is part of the data flow.

As a concrete example of the difference between the two views consider the segment "A(I)=2." In the computational view, this segment takes two inputs, A and I, and produces an output matrix which is identical to A in all elements except the I'th which is set to 2. In the data flow view, the segment has no inputs. As an output, it has only the single element A(I) which happens to be part of an array, and is set to 2.

The second approach makes it easier to understand what is going on, but it can only be used when the value of I, and hence the identity of A(I), can be determined at the time of the analysis. In particular this means that I cannot depend on the value of any datum coming from outside of the program being analyzed. When analyzing the segment above, the system would start with the computational view, and then switch to the data flow view if it discovered that it knew the value of I.

III.2 THE STRUCTURES USED TO DESCRIBE A SEGMENT

As was said above, the grand plan for a program shows how the goal of the program is decomposed, step by step, to goals achievable by primitive programs. Further, each node of the grand plan is a plan explaining how a goal is decomposed into subgoals. It is now time to look at the precise structure of a plan. To that end, the next section details the structure of behavioral descriptions. They describe the input/output behavior of segments, and form an integral part of plans.

III.2.1 BEHAVIORAL DESCRIPTIONS

A behavioral description consists of five parts: a set of input items, a set of output items, a set of prerequisites, a set of assertions, and a mapping. The prerequisites are logical conditions involving the input objects. They must be met if the segment is to behave correctly. The assertions are logical statements which are true after completion of the segment. In terms of the input and output items, the assertions say what the segment does when the prerequisites are met. Note particularly, that the definition of a behavioral description speaks of input and output items, not variables. An item is a piece of information (a number or an array or a function) which is passed between segments. A variable is just the most common data flow construct. The mapping specifies when an output item is identical to an input item.

```
Segment
        YN1 = F(XN1)
        XN1 = XN1+DX

Behavioral description of the segment
        inputs: x,inc
        prerequisites: floating numbers (x, inc)
        outputs: newx, newy, oinc
        assertions: floating numbers (newx, newy)
                newx=x+inc
                newy=F(x)
                oinc=inc
        mapping: (oinc↔inc)
```

Fig. 8: An example of a behavioral description for a two line segment. The lower case names (for example x, inc, newx) are names for items. Any similarity between these names, and the variable names in the example segment is just for the convenience of the reader. The system encodes no information in the names.

If a behavioral description accurately describes a segment, then if the inputs are supplied and the prerequisites satisfied, the outputs will appear, and the assertions will be true. In order to justify the claim that a given behavioral description is accurate, the system must look at the internal structure of the segment. The mapping is of assistance since anything which is true of an item under one name, is true under another name. The system knows, in advance, behavioral descriptions of all of the basic programs available in FORTRAN.

III.2.1.1 POINTS OF VIEW

To be useful, a behavioral description must be accurate. However, exactly what is put into a behavioral description is a function of the purpose to which the behavioral description will be put. One segment can be described from many points of view, and several segments can be described from a common point of view.

There are two key types of points of view: "intrinsic" and "extrinsic." An intrinsic behavioral

description describes a segment from an internal point of view. It references nothing external to the segment. Everything it says about the segment is true by virtue of the segment's internal workings and is independent of where, how, and why the segment is used.

An extrinsic behavioral description describes a segment from an external point of view. It references nothing internal to the segment. It describes the segment in terms of the environment of its use. An extrinsic behavioral description is only used in conjunction with other extrinsic behavioral descriptions sharing the same point of view.

III.2.1.2 RELATING BEHAVIORAL DESCRIPTIONS

Given two behavioral descriptions for a given segment, it may by desirable to show how they correspond. This is done be specifying an additional mapping which shows how the items mentioned in the two descriptions correspond.

The next figure shows an extrinsic behavioral description for the segment in the example above. A correspondence mapping is included, as part of its mapping component, which relates it to the intrinsic behavioral description of the segment in the previous figure.

```
Segment
        XN = XN1
        YN = YN1
        YN1 = F(XN1)
        XN1 = XN1+DX

Extrinsic description of the segment
        inputs: x, y, deltax
        prerequisites: floating numbers (x, y, deltax)
                0.0≤x
                0.0<deltax
        outputs: nextx, nexty, oldx, oldy
        assertions: floating numbers (nextx, nexty, oldx, oldy)
                nextx=x+deltax
                nexty=F(x)
                oldx=x
                oldy=y
                nextx>oldx
        mapping: (x↔x, deltax↔inc, nextx↔newx, nexty↔newy; oldx↔x, oldy↔y)
```

Fig. 9: This is an example of an extrinsic behavioral description of a use of the segment whose intrinsic behavioral desription is given in the last figure. Data flow connective tissue has been added to create more outputs. Similarly prerequisites have been added, yielding more complex assertions about nextx. The mapping component consists of two parts separated by a semicolon. The first part shows how names used in the extrinsic description map to names used in the intrinsic description. The second shows how names in the extrinsic description directly map together. Anything which is said of a name is true for any name it is mapped to and vice versa. Any identity between extrinsic and intrinsic names is accidental and carries no information. The mapping carries the information.

The correspondence mapping enables the system to use the fact that the intrinsic behavioral description is accurate to help show that the extrinsic behavioral description is accurate. The extrinsic behavioral description is accurate if the intrinsic behavioral description is accurate and the following four requirements are met.

First, each item in the intrinsic inputs must be mapped to by an element of the extrinsic inputs. Second, each element of the extrinsic outputs must be mapped to by an element of the intrinsic

outputs or directly from a member of the extrinsic inputs. Having an extrinsic output come directly from an extrinsic input is a common method of extending the intrinsic abilities of a segment. Third, the extrinsic prerequisites must imply the intrinsic prerequisites. If not, the segment is being incorrectly used. Finally, the extrinsic assertions must be implied by the intrinsic assertions, and the extrinsic prerequisites. If not, there is no basis for asserting them.

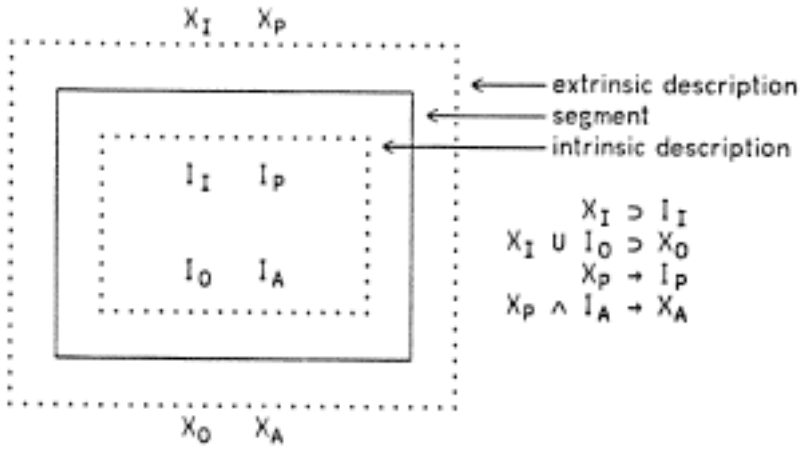


Fig. 10: This figure shows a schematic representation of the relationship between intrinsic and extrinsic descriptions of a segment. To the right, equations summarize the key relationships among the extrinsic and intrinsic inputs, outputs, prerequisites, and assertions. Throughout the rest of this paper, subscripts will be used to refer to the inputs, outputs, prerequisites, and assertions of the behavioral description of a segment. In the figure, "X" stands for the extrinsic name of the segment, and "I" for the intrinsic name. Therefore, "X_O" stands for the outputs of the extrinsic behavioral description of the segment.

III.2.2 PLANS

The plan for a segment shows how the behavior of the segment is produced through the combined effort of a set of subsegments. It consists of four parts.

Firstly, it lists extrinsic descriptions of the subsegments from a common point of view. These describe the behavior of the subsegments in the context of the outer segment. Secondly, the plan includes the intrinsic description of the segment as a whole, from the same point of view. This is the goal to be achieved. Thirdly, there is an indication of the plan type of the plan and of how the subsegments map into the components of that plan type. Finally there is a description of the teleological structure of the plan. The teleological structure contains the key information which allows the system to know why the program is the way it is.

For each plan type there is a collection of specific methods and information (see section III.3). This knowledge specifies how subsegments interact in accordance with the plan type. It shows what control and data flow are required. More importantly, it shows how the extrinsic descriptions of the subsegments are logically combined in order to yield the intrinsic description of the segment as a whole. It indicates how to go about verifying that the segment as a whole works correctly, and what some of the common bugs are in programs using the plan type.

Looking at plans from another point of view, a plan is an instantiation of a plan type. The extrinsic behavioral descriptions instantiate the subsegments. Data item naming conventions instantiate the data flow. Subsegment naming conventions specify the control flow through reference to the plan type. Finally the intrinsic behavioral description of the outer segment, together with the description of the teleological structure, instantiate the logical structure of the plan type.

The teleological structure is described using a net of two kinds of links ("purpose," and "reason"). Purpose links specify the purpose of a feature of the plan. They are intended to show

why the major design decisions were made. As a result, only the main goal directed features have purpose links. It should be noted, that a feature that does not have an avowed purpose may still be used as a justification for something. If it is not used for anything, then it is just incidental (there are many incidental features in a typical program). In the example plan, it is not just incidental that the function F produces a floating number output, however, it is not its purpose either. Its purpose is to compute the function F as required by the intrinsic description of the outer segment.

Purpose links show outputs designed to become inputs to other subsegments and/or outputs of the outer segment. They show assertions satisfying prerequisites of other subsegments and/or being used to imply assertions of the outer segment's behavioral description. They also show more global ideas, for instance, that a particular subsegment determines whether a loop should terminate. In addition, purpose links show that the inputs and prerequisites of the intrinsic behavioral description of the outer segment are designed to provide for inputs and prerequisites of subsegments.

The reason links explain why certain things can be claimed. In particular they indicate why the assertions of the intrinsic behavioral description of the outer segment can be claimed and why the prerequisites of the extrinsic behavioral descriptions of the subsegments will be satisfied. They show the key set of assertions from which another assertion can be inferred.

Taken together, the reason links are a trace of a proof of correctness for the segment. This system is unable to verify this proof because it must take many reason links on faith due to the fact that it cannot prove that they are valid.

```
Segment
        YN1 = F(XN1)
        XN1 = XN1+DX

Plan for the segment which is of type AND
(ID REA PUR) Extrinsic description of part A1 of A (F)
1   -   -          Inputs: x
2   12  -          prerequisites: floating number (x)
3   -   13         outputs: newy
4   -   -          assertions: floating number (newy)
5   -   16                 newy=F(x)
                   mapping: (newx↔arg1, newy↔return_value;)
               Extrinsic description of part A2 of A (+)
6   -   -          inputs: x, inc
7   12  -          prerequisites: floating numbers (x, inc)
8   -   13         outputs: newx
9   -   -          assertions: floating number (newx)
10  -   15                 newx=x+inc
                   mapping: (x↔arg1, inc↔arg2, newx↔return_value;)
               Intrinsic description of part A of A (the whole segment)
11  -   1,6        inputs: x, inc
12  -   2,7        prerequisites: floating numbers (x, inc)
13  -   -          outputs: newx, newy
14  4,9 -          assertions: floating numbers (newx, newy)
15  10  -                 newx=x+inc
16  5   -                 newy=F(x)
```

Fig. 11: This is an example plan for segments similar too the one dealt with in the last two figures. Note that here the names for the items do have meaning because all of the behavioral descriptions have a common point of view. If they are the same in two different descriptions then they refer to the same item. The table to the left of the figure indicates the teleological links. For example, line 8 has producing the output in line 13 as its purpose, and line 7 cites line 12 as the reason it expects to be satisfied. Section III.3.1.1 summarizes the specific knowledge for plan type AND.

If a plan is to be applied to a particular segment, to explain its operation, three things must be added to it. First, there must be an indication of what code corresponds to the segments referred to in the plan. Second, there must be a listing of what data flow connective tissue implements the data flow required by the plan type. Similarly, there must be a listing of the control flow connective tissue which implements the control flow required by the plan.

The resulting structure is referred to as an applied plan or explanation. It can be used to answer questions about how a particular segment does what it does. It should be noted that this is a second level of instantiation. The first level made explicit the structure (teleological, data flow, and control flow) of the segment. This level makes explicit the way in which the structure is implemented.

```
Segment with segmentation information
        A A1    YN1 = F(XN1)
        A A2    XN1 = XN1+DX

Applied plan for the segment (additional information only)
        data flow
                into part A1 of A (F)
                        x from outside A via variable XN1
                into part A2 of A (+)
                        x from outside A via variable XN1
                        inc from outside A via variable DX
                to outside of A
                        newy from part A1 via assignment, and variable YN1
                        newx from part A2 via assignment, and variable XN1
        control flow
                from outside A to part A1 via initial placement
                from part A1 to part A2 via sequential placement
                from part A2 to outside via final placement
```

Fig. 12: This shows the additional information which must be added to the plan in the previous figure in order to make it an applied plan, or explanation. The segmentation information is represented alongside the example program in the same was as in Fig. 5.

This system uses different methods for the understanding and manipulation of each type of plan. This allows great flexibility. This is only possible because the number of plan types is small. The next section describes the essential features of the plan types found in the programs in the IBM SSP.

III.3 THE BASIC PLAN TYPES

The following sections detail the structures of the basic plan types. The sections are grouped into catagories based on the way the plans approach decomposition of a goal. Each section follows the same pattern.

Each section starts with a schematic representation of how subsegments are combined in accordance with the plan. Arrows in the diagram indicate typical control flow (solid lines) and data flow (dashed lines). Control and data flow are not part of a plan unless it is applied to a specific program. However, the way the subsegments interact highly constrains the form the data and control flow can take. A "+" is used to mark the exit path on which a predicate asserts its assertions. It asserts the negation of its assertions on the other path.

Above and below the diagrams, equations summarize how the extrinsic inputs, outputs, prerequisites, and assertions of the subsegments combine to produce the intrinsic inputs, outputs, prerequisites, and assertions of the outer segment. As before, subscripts are used to refer to the parts of the description of a segment. Below these are explicit descriptions of the purpose (subscript PR) of each subsegment. These descriptions in conjunction with the equations indicate the teleological structure of the plan.

To the right of the diagram, equations summarize any definitions needed to understand the other equations. At the bottom of the figure, there is an example segment of FORTRAN code illustrating the plan type. Lastly, there is a section describing the plan type and indicating any points of special interest.

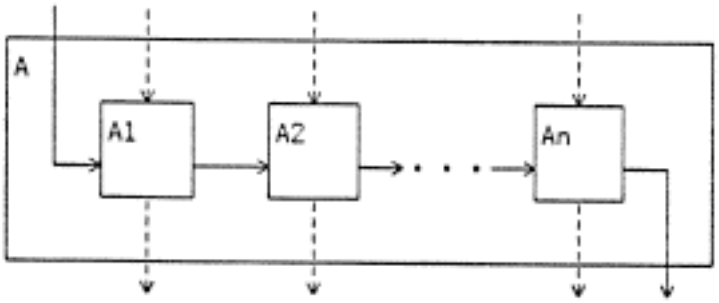
III.3.1 THE GOAL DECOMPOSITION METHOD "AND"

In this strategy, a goal is divided into pieces which can be achieved in isolation. In order to achieve the goal, each piece is achieved separately.

III.3.1.1 THE PLAN TYPE "AND"

$A_I = U_{i=1,n}\ Ai_I$

$A_P = (\wedge_{i=1,n}\ Ai_P) \wedge (\wedge_{i=1,n}\ Ai_A \neq false)$



$A_O = U_{i=1,n}\ Ai_O$

$A_A = \wedge_{i=1,n}\ Ai_A$

Ai_{PR} = contributes Ai_O to A_O and Ai_A to A_A.

```
X = SIN(Y)
A = ABS(Z)
```

Fig. 13: Schematic for, and example of, the plan type AND

This is the simplest type of plan. N subsegments (with extrinsic names Ai) are additively combined to produce the overall segment (whose intrinsic name is A). The purpose of each internal segment is to achieve its part of the overall goal. There is no a priori constraint on the order of

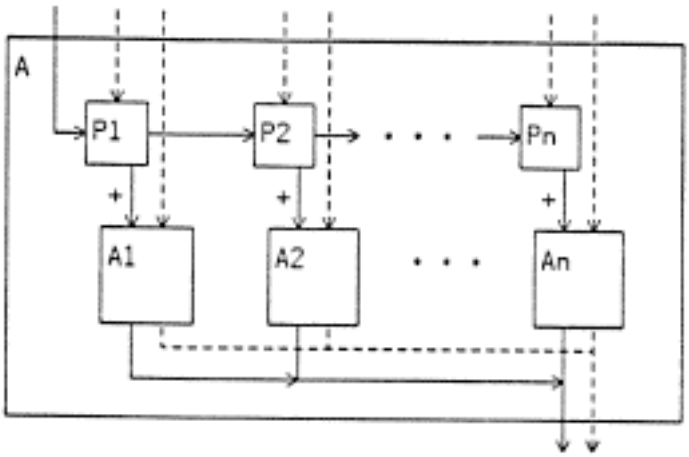
execution of the internal segments. They do not interact in any way. No data flows between them. It should be noted that the union of the assertions of the Ai should not be a contradiction. If it is, then what one subsegment is trying to achieve is incompatable with what another subsegment is trying to achieve.

III.3.2 THE GOAL DECOMPOSITION METHOD "XOR"

In this strategy, a goal is subdivided into cases. In order to achieve the goal, it is determined what type of situation exists. Based on the type, an easier goal is achieved which is equivalent in the particular situation.

III.3.2.1 THE PLAN TYPE "CASE XOR"

$A_I = U_{i=1,n}\ (Pi_I \cup Ai_I)$
$A_P = \wedge_{i=1,n}\ (Pi_P \wedge (Pi_A \rightarrow Ai_P)) \wedge (XOR_{i=1,n} Pi_A)$



$A_O = A1_O = A2_O = \ldots = An_O$
$A_A = \wedge_{i=1,n}\ (Pi_A \rightarrow Ai_A)$

Pi_{PR} = to establish whether Pi_A is true.
Ai_{PR} = when Pi_A is true, this provides the A_O and Ai_A.

```
      IF (I-1) 10,5,10
5     X = SIN(Y)
      GOTO 30
10    IF (I-2) 20,15,20
15    X = COS(Z)
      GOTO 30
20    STOP
30
```

Fig. 14: Schematic for, and example of, the plan type CASE XOR

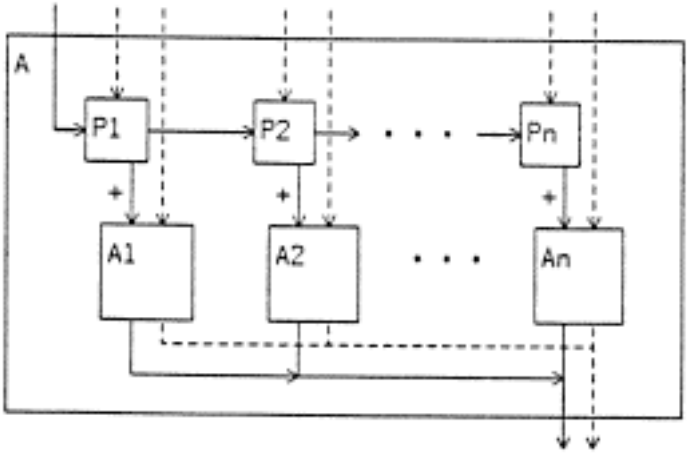
Depending on which of n situations (Pi_A) is found, one of n actions (Ai) will be performed. It must be true that one and only one of the situations occurs at a given time. This being the case, there is no restriction on the order in which the tests are made. Note that whichever test is chosen to be made last can be omitted (though this does not add to the clarity of the code).

The tests are made with predicates. In the example, the first predicate is "I-1=0". The IF converts the segment "I-1" which outputs a number into a predicate. In a given situation, the predicates, cascaded together, select the correct subsegment to use as the body of the CASE XOR. As

in the AND the subsegments (A1-An) do not interact with each other. In fact in any one execution of a CASE XOR only one of the Ai is executed.

III.3.2.2 THE PLAN TYPE "COND XOR"

$A_I = U_{i=1,n}\ (Pi_I \cup Ai_I)$
$A_P = \wedge_{i=1,n}\ (P'i_A \rightarrow (Pi_P \wedge (Pi_A \rightarrow Ai_P))) \wedge (\vee_{i=1,n} Pi_A)$



$P'i_A = \wedge_{j=1,i-1} \neg Pj_A$

$A_O = A1_O = A2_O = \ldots = An_O$
$A_A = \wedge_{i=1,n}\ ((P'i_A \wedge Pi_A) \rightarrow Ai_A)$

Pi_{PR} = to establish whether Pi_A is true.
Ai_{PR} = when $(P'i_A \wedge Pi_A)$ is true, this provides A_O and Ai_A.

```
      IF (X) 5,5,10
5     ERROR = 1
      GOTO 30
10    IF (Y/X) 15,15,20
15    ERROR = 2
      GOTO 30
20    ERROR = 0
30
```

Fig. 15: Schematic for, and example of, the plan type COND XOR

The COND XOR is a variant of the CASE XOR. It is included here because it is extremely common and because bugs often arise due to a confusion between the two. In the COND XOR, the order of execution of the tests is not free, it is essential.

In a serial computer, the tests must be made in some order. The COND XOR takes advantage of this fact to gain two benefits. First the situation in which a subsegment Ai is executed is $(P'i_A \wedge Pi_A)$ not just (Pi_A) (actually this was true with the CASE XOR as well, but since $((XOR_{i=1,n} Pi_A) \rightarrow (Pi_A \rightarrow P'i_A))$ it was not useful). This is useful here because the alternative situations needed to subdivide a goal often have this kind of form. In addition, the prerequisite of the COND XOR requires only that the OR (rather than the XOR) of the predicates Pi_A be true. This is, in general, more in line with the way that people think. In the example, the three situations are $X \leq 0$, $X > 0 \wedge Y/X \leq 0$, and $X > 0 \wedge Y/X > 0$. Using the COND XOR, rather than an equivalent CASE XOR, saves duplication of programming effort.

The second gain is that the prerequisites of a COND XOR are often simpler than in an equivalent CASE XOR. In the example, because the order of the tests is fixed, the prerequisite of the second test (that $X \neq 0$) does not need to be a prerequisite of the whole COND XOR. Note that, as in the

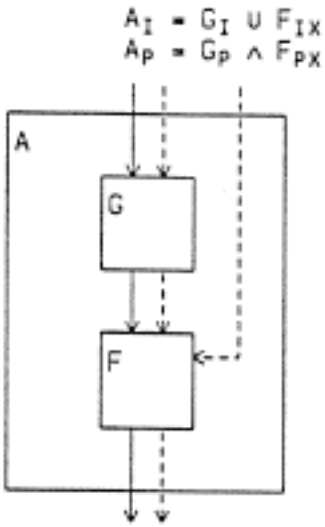
example, the last predicate of a COND XOR is often "true" (needing no code to be implemented) which recognizes the situation "otherwise."

One of the most common bugs associated with XORs is due to the fact that programmers often implement a COND XOR when they have a CASE XOR in mind. In the example, the programmer might have been thinking that whenever X≤0 then error=1 and whenever Y/X≤0 then error=2. It is easy not to notice that this makes no sense since the alternatives are not mutually exclusive. What happens when both alternatives are true? The program makes an arbitrary decision setting error to 1. This may cause a problem later if some other program assumes that error≠2 implies that y/x>0.

III.3.3 THE GOAL DECOMPOSITION METHOD "COMP"

In this strategy (composition), a goal is achieved one step at a time. A subgoal is chosen so that after the subgoal is achieved it is easier to achieve the desired goal.

III.3.3.1 THE PLAN TYPE "COMP"



G_{PR} = to satisfy F_{PI}, to provide F_{II}, and to contribute G_{AX} to A_A.
F_{PR} = to provide A_O and to contribute F_A to A_A.

Z = ABS(X)
Y = LOG(Z)

Fig. 16: Schematic for, and example of, the plan type COMP

The subsegment F performs a calculation based on external data and on the ouputs of the subsegment G. The inputs and prerequisites of F are divided into two sets external (F_{IX} and F_{PX}) and internal (F_{II} and F_{PI}). The internal requirements are satisfied by the outputs and assertions of G.

The vertical splitting of the goal has two main benefits. First, the prerequisites of the whole segment are often simpler than the prerequisites of F, since some of the later are satisfied by the assertions of G. More importantly, the assertions of the outer segment compose the assertions of F and G. In the example, G makes the assertion Z=ABS(X) and F makes the assertion Y=LOG(Z). Often, substitution is used to eliminate references to the outputs of G in the assertions of A. In the example, this would yield Y=LOG(ABS(X)) as the assertion of A.

An important special case of the plan type COMP occurs when A_A equals F_A and F_A does not

mention the outputs of G. In this case (referred to as a PREP COMP), the only function of G is to satisfy prerequisites of F.

This plan type is not common in numerical programming. However, it does occur in the program RK1. The following example from the blocks world illustrates the plan type.

```
CLEARTOP(A)
PUTON(A,B)
```

Fig. 17: An example of the plan type PREP COMP

The goal is to put A on B (B is assumed to already have a clear top). G which is CLEARTOP(A) contributes to this goal only by making F which is PUTON(A,B) applicable.

III.3.4 LOOPS

The previous sections have described three methods for transforming one goal into a set of easier subgoals so that achieving the subgoals achieves the goal. The plan types described above are limited in that, even in combination, they can only be used to implement a goal decomposition method when the set of subgoals can be bounded in size at the time the program is written. Loops can be used to implement unbounded, but finite, sets of subgoals which are sufficiently repetitive.

Loops are not a different goal decomposition method. They are a different implementation method. It should be noted that loops are also often used to implement bounded sets of subgoals which have repetitive structure.

There are two ways to approach the description of the plan type LOOP. One could start with a method for describing unbounded but finite sets of subgoals and show how they can be implemented. Alternately, one could start with the phenomenon of a loop and show how it can be harnessed to do useful work. Each way highlights interesting aspects of the problem.

Consider a finite but a priori unbounded set of subgoals which can be put in one to one correspondence with an initial subset of the positive integers so that achieving the subgoals in the natural order of the corresponding integers achieves the overall goal. Let G[i] represent the subgoal associated with the integer i (square brackets will be used exclusively to mark items associated with the i'th iteration of a loop). The set of subgoals is then {G[1], ... ,G[i], ... , G[n]}, where n is the unknown number of subgoals.

Suppose that there is a set of computations {B[1], ... , B[i], ...} such that if {G[1], ... , G[i-1]} have already been achieved, then executing B[i] achieves G[i]. Further, suppose that there is a set of predicates {T[0], ... , T[i], ...} such that if {G[1], ... , G[i]} have already been achieved, then executing T[i] will determine whether the overall goal has been achieved, i.e. whether n=i. Given this, the following unbounded computation would achieve the overall goal and terminate when it has been achieved.

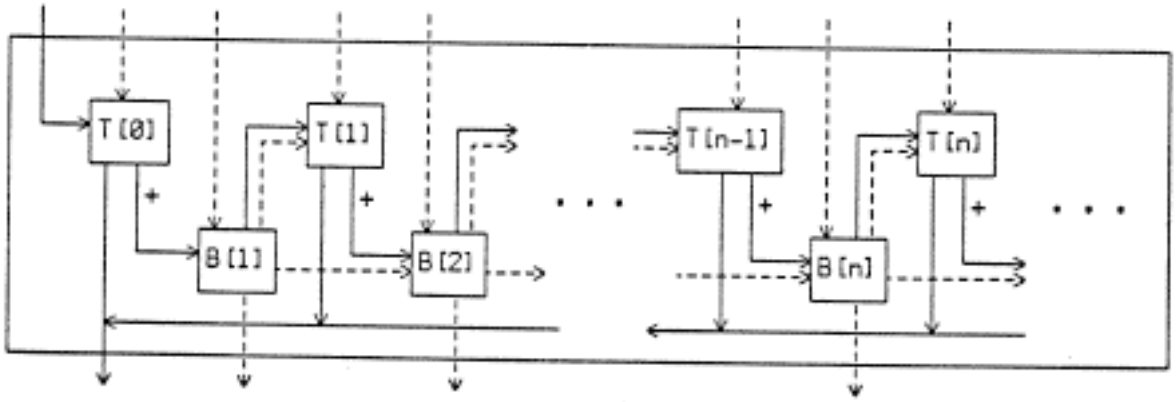


Fig. 18: Schematic for the plan for an unbounded computation achieving the {G[i]}

The sequence of computations cannot be implemented in the way the above figure suggests. If each segment was implemented separately the resulting program would be infinite, since n is not known in advance.

A loop has the property that it can produce an unbounded sequence of computations from a finite amount of code.

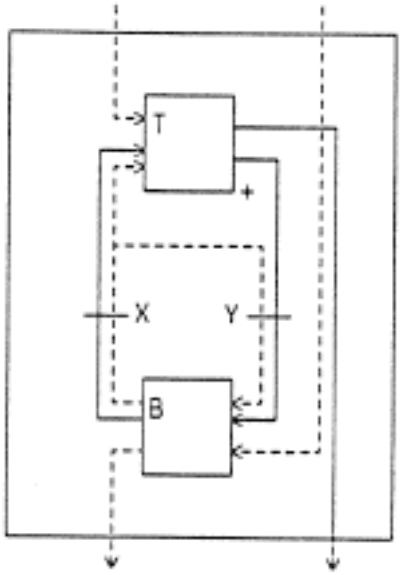


Fig. 19: a loop

The above plan schematic has loops in control and data flow. If it were running, it could clearly keep running for an unbounded time, alternately executing B and T. Further, since T is a predicate, it might even terminate. There is a problem however, how would it start? It appears that the execution of T must always precede the execution of B and that the execution of B must always precede the execution of T.

If the execution of the loop were already at either point X or point Y, then there would be no problem. A loop may be initiated by mimicking every relevant feature of the state of the world at X or at Y and then proceeding as if the loop had always been running. What features are relevant depends on the nature of T and B. Clearly, any data items which will be used as inputs must be created, and the prerequisites of T and B must be satisfied.

The notion of mimicry is a powerful one in its own right. It is brought into play whenever two data or control flow paths join together to become one. The joining causes the information about the origin of the paths, which was inherent in their separateness, to be lost. On the other hand, when paths diverge, as from a predicate, information can be encoded in the separateness of the paths.

The plan types XOR, COMP, and LOOP approach mimicry and path convergence in different ways. In an XOR (see section III.3.2) the data and control flow paths join together at the bottom. The output assertions explicitly mention this fact. In effect they say that one of these n things happened and it can be decided which one by determining which predicate was true.

In a COMP (see section III.3.3) there is not actually any joining of paths. However, imagining that there is potentially a second control flow and data flow path directly to F from the outside provides a convenient way of thinking about what is happening. If everything was set for the execution of F, then the direct path could be taken. G creates the required conditions, thereby mimicking the conditions on the direct paths to F. Note that, in general, F cannot tell whether control comes from G or directly from the outside.

This notion can be extended. Whenever the relevant state is indistinguishable in two places and the same subsequent calculation is desired, then a GOTO can be done from one place to the other.

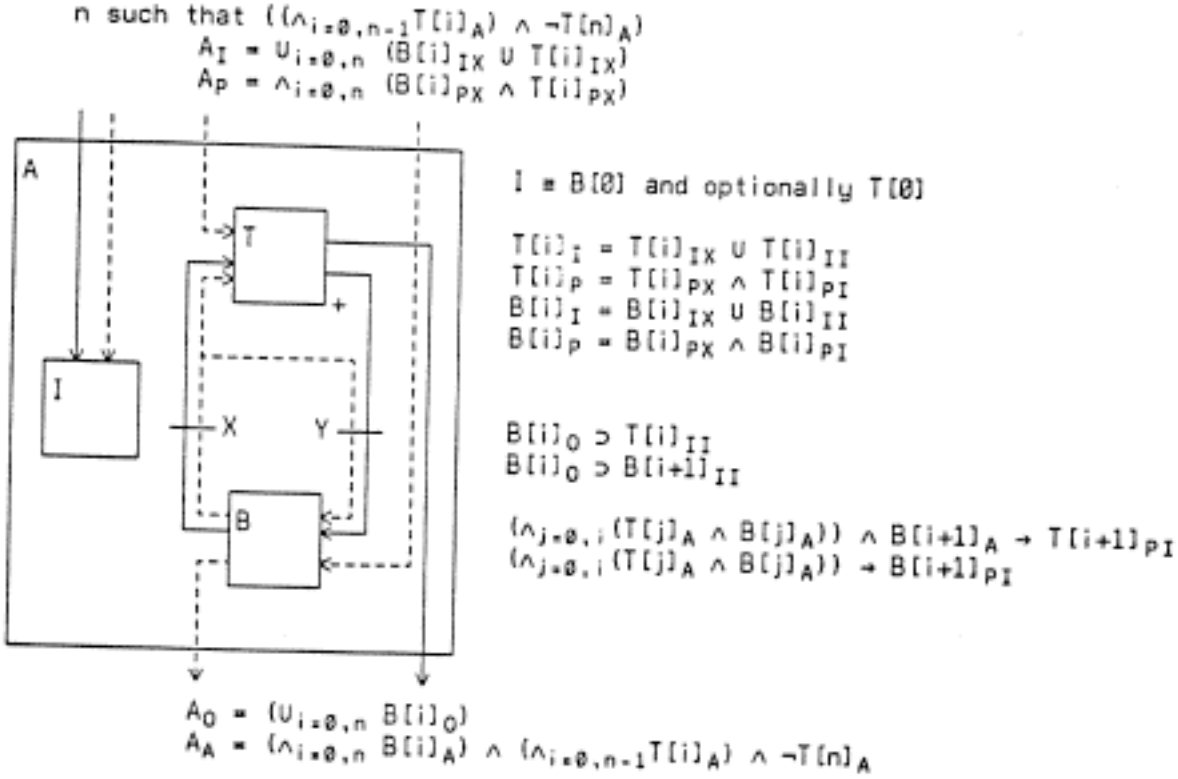
This causes a joining of data and control flow paths. If the computations beyond the two points are not directed toward the same goals, but just happen to be functionally equivalent, then the joining is just a case of factoring (see section III.1.3.3) and is liable to be confusing programming.

When the goals are the same, the GOTO can be a very useful way to look at a problem. It could have been a plan type on its own. However, the programs in the SSP tend to follow the ideas of structured programming. As a result, the notion of GOTOs and mimicry can be restricted to stereotyped positions in certain plan types. This makes it easier to deal with GOTOs, just as a similar restriction makes it easier to deal with predicates.

Returning to LOOPs, suppose that the loop above is started at point X and that T and B have the following properties. The first time T is executed, it computes T[0]. The second time it is executed it computes T[1], and so on. The first time B is executed, it computes B[1]. The second time B is executed, it computes B[2], and so on. If this is true, the loop will perform exactly the same computation as the unbounded program.

In addition, the loop will take up finite space if T and B do. If this is to be the case, the {T[i]} and the {B[i]} must be sufficiently repetitive in structure so that they can be implemented by finite programs. It should be noted that T and B will most likely contain computation whose sole purpose is to make it possible for them to determine which iteration is the current one so that they know what to compute. In the unbounded program, this information is contained in the flow of control. In a loop, it must be encoded in other ways.

III.3.4.1 THE PLAN TYPE "LOOP"




```
    X(0) = INIT
    DO 10 I=1,INFINITY
    X(I) = F(I,X(I-1),Y(I))
    IF (T(I,X(I),Z(I)) 20,10,20
10  CONTINUE
20  ...
```

```
    X = 1
10  X = X-((X*X-10)/(2*X))
    IF (ABS(X*X-10)-1.0E-10) 20,20,10
20  ...
```

Fig. 20: Schematic for, and examples of, the plan type LOOP

T and B are as described above. I is an additional segment which performs the mimicry in order to start the loop at either X or Y. In the equations in the figure, I is referred to as B[0] (this is done for notational convenience). If I mimicks the situation at point Y, then it in addition mimicks what is referred to as T[0]. Note that this can (and often does) cause trouble since B[1] is always executed if the LOOP is started at point Y. There is no way to differentiate between the case where only the goal G[1] should be achieved and the case where no subgoals should be achieved.

The equations showing how the subsegments interact in order to create the behavior of a LOOP are complex. Fortunately, most LOOPs do not take advantage of the full complexity. In the figure, two example segments are given. The first, which has more or less full complexity, is artificial and could be looked upon as a schema of a LOOP. The second, which finds the square root of 10, is much simpler. Further, LOOPs can be broken up in order to isolate different areas of complexity.

Probably the greatest area of complexity is determining when, if ever, a given LOOP will terminate. The quantity "n," which represents the termination point, appears in every equation and is central to the understanding of a LOOP. The next two sections show how the problem of working with

n can be split off from the rest of the LOOP. Once isolated n often turns out to be relatively easy to understand.

There are other areas where simplification can occur. These areas could be looked at as features which a given LOOP might or might not have. The expressions for the inputs and prerequisites are often simplified. B and/or T may not have any external inputs or prerequisites. Failing this, the expressions for A_I and/or A_P may not depend on n (this is the case in the second example segment).

Very often, the only outputs are from the last step. Similarly, the assertions may only depend on the last step. It should be noted that picking just the right assertions for T and B so that the entire action of the LOOP is summarized at each step is an art. Put another way, it will often be very hard to prove that a convenient form for the assertions of A follows from a convenient form for the assertions of T and B. When faced by a difficult proof, this system will just ask the user whether it is possible. If he says that it is, then the system will trust him. However, it will remember that the assertion may be shown false at a later time.

III.3.4.2 THE PLAN TYPE "ENUMERATION LOOP"

If everything is removed from a LOOP except for the calculation of n, then the remaining LOOP is an ENUMERATION LOOP. The process described in the next section shows how additional computation can be added into a LOOP.

```
    DO 20 I=1,100
20  CONTINUE
```

```
    X = 0.0
10  X = G(X)
    IF (X) 10,20,10
20  ...
```

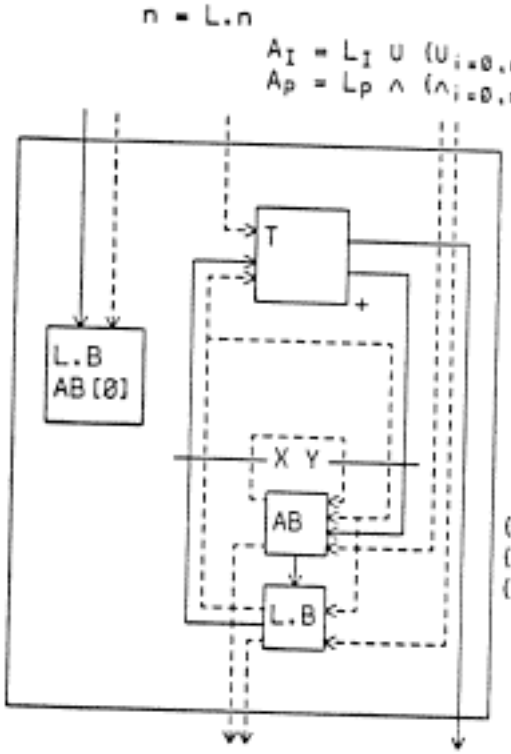
Fig. 21: Examples of the plan type ENUMERATION LOOP

The schematic for this type of LOOP is identical to that for the general LOOP except the it does not produce any output. It just cycles through a sequence of states. In doing this it defines an ordered set of situations. The only difficulty in understanding an ENUMERATION LOOP is determining when it will terminate. Since an ENUMERATION LOOP theoretically can be as complex as any LOOP, it may not be easy to understand. However, in general when the parts of a LOOP which do not contribute to the calculation of n are stripped away, the ENUMERATION LOOP which remains is easy to understand. Though this type of LOOP seldom appears, as is, in a program, it is very important as a basis for understanding more complex LOOPs which are built up on the basic series of states it enumerates.

If an ENUMERATION LOOP is slightly extended so that it returns $B[n]_0$ as its output, then it is referred to as a SEARCH LOOP.

III.3.4.3 THE PLAN TYPE "AUGMENTED LOOP"

This plan type combines a loop L and an additional body AB computing {AB[1], ... , AB[i], ...} to form a more complex LOOP. In the figure, X.Y stands for part Y of the segment X.



T = L.T
B = an AND of L.B and AB
I = an AND of L.I and AB[0]

$L.B[i]_O \supset T[i]_{II}$
$L.B[i]_O \supset L.B[i+1]_{II}$
$L.B[i]_O \cup AB[i]_O \supset AB[i+1]_{II}$

$(\wedge_{j=0,i}(T[j]_A \wedge L.B[j]_A)) \wedge L.B[i+1]_A \rightarrow T[i+1]_{PI}$
$(\wedge_{j=0,i}(T[j]_A \wedge L.B[j]_A)) \rightarrow L.B[i+1]_{PI}$
$(\wedge_{j=0,i}(T[j]_A \wedge L.B[j]_A \wedge AB[j]_A)) \rightarrow AB[i+1]_{PI}$

$A_O = L_O \cup (\cup_{i=0,n} AB[i]_O)$
$A_A = L_A \wedge (\wedge_{i=0,n} AB[i]_A)$

I_{PR} = to initiate the looping.
$T[i]_{PR}$ = to determine if the computation should stop i.e. if n=i.
$L.B[i]_{PR}$ = to perform the calculation for LOOP L.
$AB[i]_{PR}$ = to perform the added steps of calculation.

```
    DO 10 I=1,10                    Z = 1.0
10  X(I) = Y(I)*Z(I)                DO 10 I=1,10
                                10  Z = Z*X
```

Fig. 22: Schematic for, and examples of, the plan type AUGMENTED LOOP

Note that this is a three level plan. AB is added to L.B (a subsegment of the LOOP L) to form a new body. There is no restriction on the way AB and L.B interact except that no data can flow from AB to L.B or to T. I may also be modified by the addition of AB[0] in order to initialize the actions of AB.

The most important thing here is that the addition of AB to L does not effect L's termination. If L was understood, then the new loop A is easy to understand. Any segment which can be explained by this plan can also be explained by the general LOOP plan. Using this plan is more advantageous because it develops a better understanding of the whole segment by looking at the internal structure of segment B.

There are two major subcases of this plan type. The division is based on whether AB uses feedback or not. The examples illustrate the two types: AND AUGMENTATION and COMP

AUGMENTATION. In both cases, a computation is performed, taking advantage of the sequence of states which is set up by the LOOP L, which is usually an ENUMERATION LOOP (as in the examples).

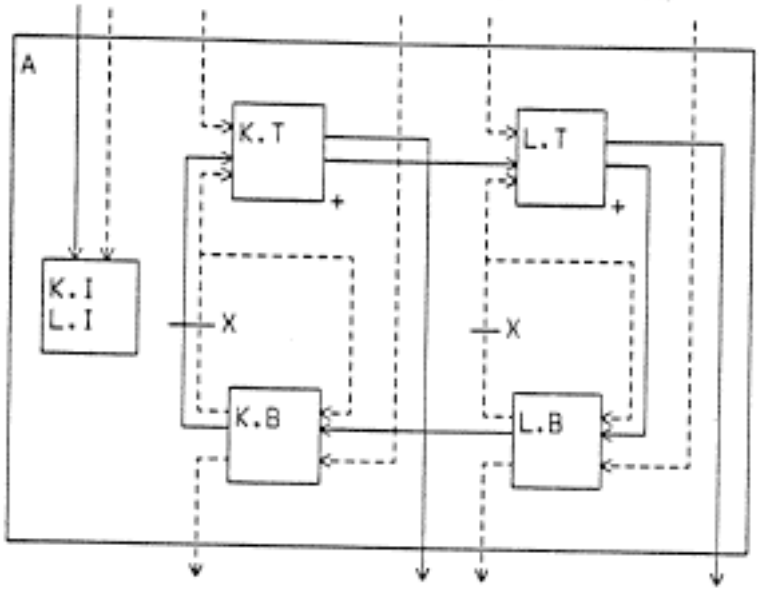
III.3.4.4 THE PLAN TYPE "INTERLEAVED LOOP"

This plan type combines two LOOPs, K and L, so that they are computed in synchrony. The combination terminates as soon as either one terminates.

$n = MIN(K.n, L.n)$

$A_I = \cup_{i=0,n} (K.B[i]_{IX} \cup K.T[i]_{IX} \cup L.B[i]_{IX} \cup L.T[i]_{IX})$

$A_P = \wedge_{i=0,n} (K.B[i]_{PX} \wedge K.T[i]_{PX} \wedge L.B[i]_{PX} \wedge L.T[i]_{PX})$



I = K.I ∪ L.I
T = K.T OR L.T
B = K.B ∪ L.B

each LOOP operates separately

$A_O = (\cup_{i=0,n} (K.B[i]_O \cup L.B[i]_O)$

$A_A = (\wedge_{i=0,n} (K.B[i]_A \wedge L.B[i]_A))$
$\wedge (\wedge_{i=0,n-1} (K.T[i]_A \wedge L.T[i]_A))$
$\wedge (\neg K.T[n]_A \vee \neg L.T[n]_A)$

I_{PR} = to initiate the looping.
$L.T[i]_{PR}$ = to see if the computation should stop due to LOOP L.
$K.T[i]_{PR}$ = to see if the computation should stop due to LOOP K.
$L.B[i]_{PR}$ = to perform the steps of the calculation for LOOP L.
$K.B[i]_{PR}$ = to perform the steps of the calculation for LOOP K.

```
      X = 0.0
      DO 1=1,10
      X = F(X)
      IF (X) 10,20,10
10    CONTINUE
20    ...
```

Fig. 23: Schematic for, and example of, the plan type INTERLEAVED LOOP

This is also a three level plan; K.T, L.T, K.B and L.B are executed in any order in a ring, with mimicry starting the LOOP at any of four points. The key requirement is that no data flows between the two subLOOPs. Their only interaction is that when one terminates, the other is artificially stopped. It should be noted that when one of the subLOOPs (say K) terminates, then the whole LOOP

(A) acts in all respects just like that subLOOP (K) with some additional computation from the partial execution of the other subLOOP (L). In particular, the outputs are the outputs of K with a subset of the outputs of L added on.

Like mimicry, interleaving is a powerful idea in its own right and could be a separate plan type. Interleaving is essentially a way to simulate parallel processing. Here it has been restricted to applying only to LOOPs because it is not interesting when applied to the other plan types discussed.

One of the most important attributes of this plan type is that it can be shown to terminate even if only one of the two LOOPs K and L can be shown to terminate. As a result of this, this plan type is often used to bound the execution of a possibly non-terminating LOOP with a simple ENUMERATION LOOP (see the example).

The other major use of this plan type is based on considering that it computes either K or L (whichever completes first). Here first is defined in terms of the sequence of states produced by the LOOPs. This is done in a situation where, for instance, the results of L are not desired (or perhaps are not computable) if K terminates first. The program RK1 contains an example of this.

In conclusion, it should be noted that control exits an INTERLEAVED LOOP in two different places depending on which test terminates the LOOP. In a grand plan this feature is expressed by considering that the interleaving process joins two LOOPs, each of which is the initial component of a complex computation. If the INTERLEAVED LOOP terminates due to test K.T, then execution continues only in the computation LOOP K is the initial component of.

The splitting of the flow of control on exit from an INTERLEAVED LOOP could have been eliminated if a component were added to all LOOPs which was executed after the test had terminated the looping. This was not done, because it was felt that composing this exit segment on the output of every LOOP would obscure the basic nature of LOOPS.

III.4 DETERMINING THE DESCRIPTION OF A PROGRAM

This section gives an indication of how the system can develop an understanding of a program in terms of the descriptive structures defined in section III.2. Given an understanding of a program, Section II gives an indication of how tasks are performed by the system. Basically, the system either just reports out parts of the descriptive structures, or asks itself a series of questions and performs some minor deductions. The descriptive structures were specifically designed to make the tasks described in section II easy.

Starting from the text of a program, including annotation, an understanding is developed using several types of knowledge. The system has complete knowledge of the basic facts about FORTRAN. This includes knowledge of the specific control flow and data flow constructs available, and knowledge of the basic programs available. The system also has some basic knowledge about mathematics. However, it should be noted that, this system does not try to understand the mathematical theorems implemented by a program, but only how the program implements the theorems. Finally, the system has knowledge of what plan types are used in the programs in the IBM SSP.

The understanding process is illustrated by a discussion of the program CONVT which is shown in the next figure.

```
      C
      C     PURPOSE
      C        CONVERT NUMBERS FROM SINGLE PRECISION TO DOUBLE PRECISION
      C        OR FROM DOUBLE PRECISION TO SINGLE PRECISION.
      C     DESCRIPTION OF PARAMETERS
      C        N    - NUMBER OF ROWS IN MATRICES S AND D.
      C        M    - NUMBER OF COLUMNS IN MATRICES S AND D.
      C        MODE - CODE INDICATING TYPE OF CONVERSION
      C                  1 - FROM SINGLE PRECISION TO DOUBLE PRECISION
      C                  2 - FROM DOUBLE PRECISION TO SINGLE PRECISION
      C        S    - IF MODE=1, THIS MATRIX CONTAINS SINGLE PRECISION
      C               NUMBERS AS INPUT.  IF MODE=2, IT CONTAINS SINGLE
      C               PRECISION NUMBERS AS OUTPUT.  THE SIZE OF MATRIX S
      C               IS N BY M.
      C        D    - IF MODE=1, THIS MATRIX CONTAINS DOUBLE PRECISION
      C               NUMBERS AS OUTPUT.  IF MODE=2, IT CONTAINS DOUBLE
      C               PRECISION NUMBERS AS INPUT.  THE SIZE OF MATRIX D IS
      C               N BY M.
      C        MS   - ONE DIGIT NUMBER FOR STORAGE MODE OF MATRIX
      C                  0 - GENERAL
      C                  1 - SYMMETRIC
      C                  2 - DIAGONAL
      C     REMARKS
      C        MATRIX D CANNOT BE IN THE SAME LOCATION AS MATRIX S.
      C        MATRIX D MUST BE DEFINED BY A DOUBLE PRECISION STATEMENT IN
      C        THE CALLING PROGRAM.
      C     METHOD
      C        ACCORDING TO THE TYPE OF CONVERSION INDICATED IN MODE, THIS
      C        SUBROUTINE COPIES NUMBERS FROM MATRIX S TO MATRIX D OR FROM
      C        MATRIX D TO MATRIX S.
      C
            SUBROUTINE CONVT (N,M,MODE,S,D,MS)
            DIMENSION S(1),D(1)
            DOUBLE PRECISION D
      C       FIND STORAGE MODE OF MATRIX AND NUMBER OF DATA POINTS
            IF (MS-1) 2, 4, 6
          2 NM=N*M
            GO TO 8
          4 NM=((N+1)*N)/2
            GO TO 8
          6 NM=N
      C       TEST TYPE OF CONVERSION
          8 IF (MODE-1) 10, 10, 20
      C       SINGLE PRECISION TO DOUBLE PRECISION
         10 DO 15 L=1,NM
         15 D(L)=S(L)
            GO TO 30
      C       DOUBLE PRECISION TO SINGLE PRECISION
         20 DO 25 L=1,NM
         25 S(L)=D(L)
      C
         30 RETURN
            END
```

Fig. 24: The subroutine CONVT

The first step taken by the system in order to analyze a program is to divide the program into control flow, data flow, basic program fragments, and comments.

```
1-31  are comments and appear in the preceding figure
31            subroutine convt (n,m,mode,s,d,ms)
32            dimension s(1),d(1)
33            double precision d
34    c           find storage mode of matrix and number of data points
35            IF (ms-1) 2, 4, 6
36          2 nm=n*m
37            GO TO 8
38          4 nm=((n+1)*n)/2
39            GO TO 8
40          6 nm=n
41    c           test type of conversion
42          8 IF (mode-1) 10, 10, 20
43    c           single precision to double precision
44         10 DO 15 I=1,nm
45         15 d(I)=s(I)
46            GO TO 30
47    c           double precision to single precision
48         20 DO 25 I=1,nm
49         25 s(I)=d(I)
50    c
51         30 RETURN
52            end
```

Fig. 25: This shows the subroutine CONVT printed in four different styles of type. The styles of type identify each part of the program as either:
1) *FLOW OF CONTROL CONNECTIVE TISSUE*
2) data flow connective tissue
3) a basic program fragment
4) c a comment.

Modulo certain transformations (see sections III.1.3.5 and III.1.3.6), this division can be done purely syntactically. For convenience, entire expressions like "((N+1)*N)/2" have been taken as basic programs in the example. The system will actually only consider functions like "+", "-", "FLOAT", array indexing, etc. to be primitive. However, it probably will treat expressions in a special way since they are particularly easy to understand.

III.4.1 CONTROL FLOW

The control flow of a FORTRAN program can be completely analyzed by looking at the text of the program, without any elaborate reasoning. Programs such as optimizing compilers currently perform such analyses. The next figure is a control flow diagram for the program CONVT. This diagram is a graphic representation of what the system knows about the control flow of the program.

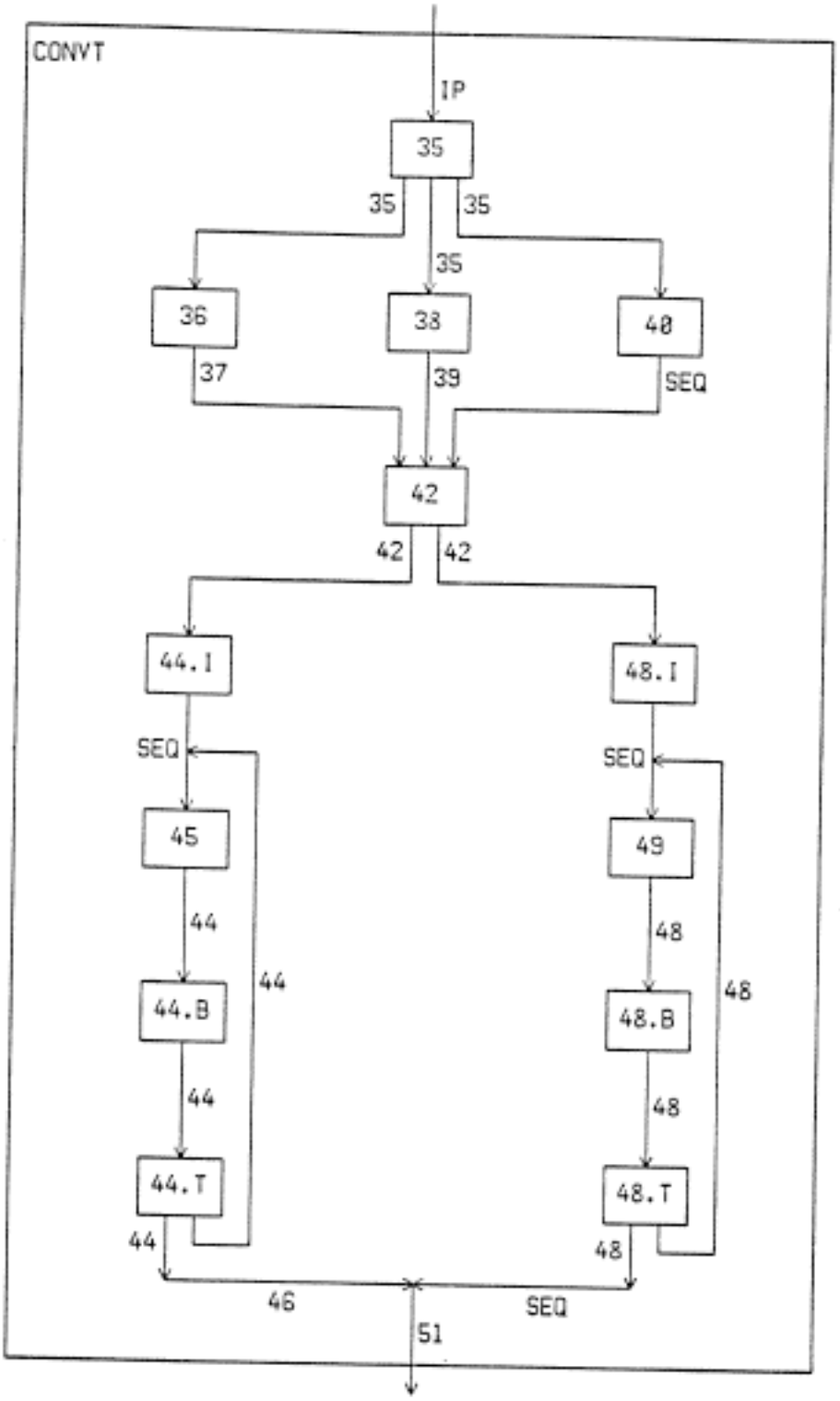


Fig 26: The control flow in the program CONVT.
SEQ=SEQUENTIAL PLACEMENT, IP=INITIAL PLACEMENT.

The diagram is a directed graph. Each arc connects one node to one other node. It represents the flow of control between the nodes. Forking of control flow arcs is only reasonable in an environment where there is asynchronous processing. The label on an arc indicates how it is implemented. This label is either a line number or a direct explanation. If it is a line number, the control flow construct on that line implements the control flow indicated by the arc. The possible direct explanations include "SEQUENTIAL PLACEMENT" and "INITIAL PLACEMENT." Both of these refer to situations where the arrangement of the statements in the program governs the flow of control and there is no explicit flow of control construct to point to.

Nodes are labeled with an indication of the activity taking place at the node. A line number indicates the basic program and/or data flow which takes place in the control environment associated with the node. For example, the actions of line 36 (computing N*M and assigning the result to NM) are computed only when MS<1. If a node is associated with only a part of a line, then it is labeled with the extrinsic name of that part of the line. For example, the label "44.B" on a node indicates that only the body, "L=L+1", of the ENUMERATION LOOP, embodied in line 44, is associated with the node. A node at which nothing happens is not represented as a box. It is just a point at which an arc terminates and another begins.

If more that one arc leaves a node, then that node must be a predicate. For example, the nodes labeled 35, 42, 44.T, and 48.T are predicates. Several arcs can enter a node. This has no extraordinary significance.

The nodes are associated with the lowest level segmentation of the program. In the figure, nodes have been selected in accordance with a line oriented selection of basic programs. The system would actually produce a more complex diagram in which no reasoning was required during the node selection process. For clarity, the figure superimposes a higher level of segmentation on the system's diagram (see the section on segmentation below). In order to avoid making any decisions about segmentation while analyzing the control flow, the system puts in as many nodes as there are distinct control environments.

III.4.2 DATA FLOW

The data flow for a FORTRAN program can also be completely analyzed in a straightforword manner. The next figure is a diagram, similar to the control flow diagram, representing the system's knowledge of the data flow in the program CONVT.

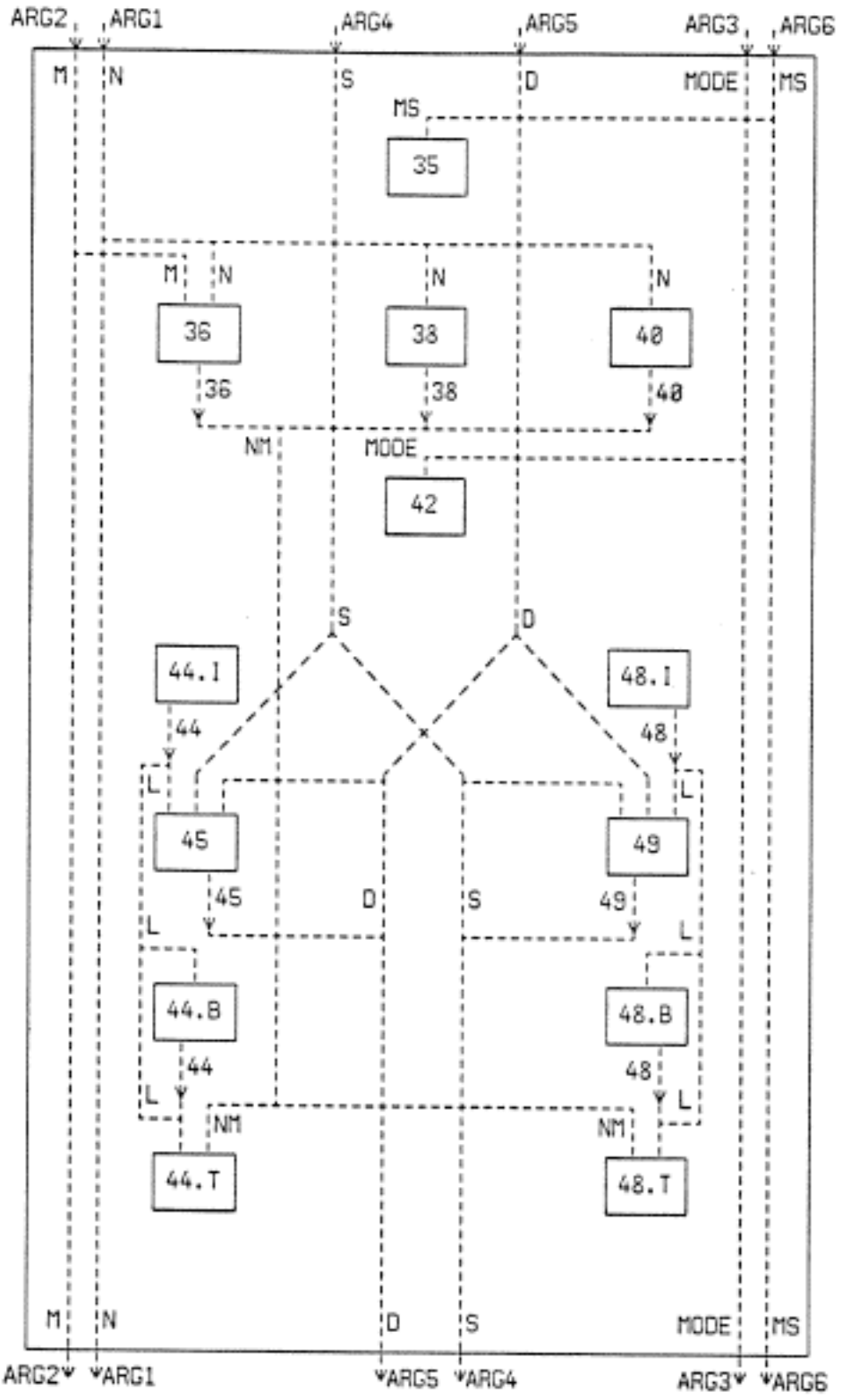


Fig. 27: The data flow in the program CONVT

Though some data flow constructs (such as assignment) are directional, many (such as variables and subroutine arguments) are not. These non-directional constructs merely maintain the same value at several points. The control flow imposes a directionality on them. The label on an arc indicates how the data flow it represents is implemented. The label is either a line number of a data flow construct, a variable name, or a direct explanation. If it is a variable name, then that variable implements the data flow. Direct explanations are used to indicate phenomena which cannot be easily isolated to one item in the program. The only one used in the figure is "ARGn" which indicates that the data flow into or out of the program was achieved through subroutine argument position n.

Unlike control flow arcs, data flow arcs can fork arbitrarily. For example, the arc implemented by variable NM connects the assignment statements leaving nodes 36, 38, and 40 with the nodes 44.T, and 48.T. However, a data flow path can actually only carry one datum at a time. As a result, a datum which is delivered to the node labeled 44.T, in a given execution of the node, actually originates at just one of the nodes 36, 38, or 40. Many arcs can enter or leave a node. This just indicates that several data items are entering or leaving the node.

The data flow nodes are selected by the system in the same way as control flow nodes are. Namely, one for every data flow environment. As before, the figure shows a higher level of segmentation.

III.4.3 BASIC PROGRAM FRAGMENTS

There are only a limited number of basic programs available in FORTRAN. The system has complete intrinsic descriptions of each one as part of its specific knowledge about FORTRAN. These descriptions embody the lowest level of detail understood by the system.

III.4.4 ANNOTATION

The information described in the previous three sections gives a complete picture of the program, at the finest level of detail. The system uses this information to build up a description of the program at intermediate levels of structure up to an intrinsic description of the program as a whole. The use of annotation of the program is essential in this process. Comments and their use will be discussed in all of the following sections.

It should be noted that this system does not try to do any natural language processing. As a result, the comments cannot be used in the exact form they take in the programs in the SSP. It has not been decided what form the comments will take. The next figure offers one suggestion.

A very important question is how much annotation will be required in order for the system to develop an understanding of a program. It appears that though the comments on the programs in the SSP will have to be recast in some other form, they contain sufficient information for understanding. As a result, the amount of annotation needed should not be greater than the amount already present on the actual programs.

```
C partial intrinsic description of CONVT
C         inputs: n, m, mode, {s(i)}, {d(i)}, ms
C         prerequisites: floating numbers ({s(i)})
C                        double precision numbers ({d(i)})
C                        integer numbers (n, m, mode, ms)
C                        matrices ({s(i)}, {d(i)})
C                        n=NUMBER_OF_ROWS({s(i)})=NUMBER_OF_ROWS({d(i)})
C                        m=NUMBER_OF_COLUMNS({s(i)})=NUMBER_OF_COLUMNS({d(i)})
C                        ms=STORAGE_MODE({s(i)})=STORAGE_MODE({d(i)})
C                        mode ∈ {1, 2}
C         outputs: on, om, omode, {os(i)}, {od(i)}, oms
C         assertions: floating numbers ({os(i)})
C                        double precision numbers ({od(i)})
C                        integer numbers (on, om, omode, oms)
C                        matrices({os(i)}, {od(i)})
C                        mode=1 → {od(i)}=DBLE({s(i)}) COMPONENTWISE
C                        mode=2 → {os(i)}=SNGL({d(i)}) COMPONENTWISE
C
31            SUBROUTINE CONVT (N,M,MODE,S,D,MS)
32            DIMENSION S(1),D(1)
33            DOUBLE PRECISION D
C
C the next segment is of plan type XOR
C partial extrinsic behavioral description of the next segment
C         assertions: OUTPUT=NUMBER_OF_ELEMENTS({s(i)})=NUMBER_OF_ELEMENTS({d(i)})
C
35                    IF (MS-1) 2, 4, 6
36          2         NM=N*M
37                    GO TO 8
38          4         NM=((N+1)*N)/2
39                    GO TO 8
40          6         NM=N
C
C the next segment is of plan type XOR
C
42          8         IF (MODE-1) 10, 10, 20
C
C partial extrinsic behavioral description of the following segment
C         assertions: {od(i)}=DBLE({s(i)}) COMPONENTWISE
C
44         10               DO 15 L=1,NM
45         15               D(L)=S(L)
46                        GO TO 30
C
C partial extrinsic behavioral description of the following segment
C         assertions: {os(i)}=SNGL({d(i)}) COMPONENTWISE
C
48         20               DO 25 L=1,NM
49         25               S(L)=D(L)
51         30 RETURN
52            END
```

Fig. 28: The annotation on the program CONVT translated to a form understandable by the system. Indentation indicates segmentation.

All of the information in the above figure comes directly from the comments on the actual program. This section will discuss the partial intrinsic behavioral description for CONVT which corresponds to the header comment, lines 1-30, together with lines 31-33. It is interesting to note that lines 32 and 33 could be looked at as comments which have been put in a form which the FORTRAN compiler can understand and act on. The following sections will discuss the internal comments.

Consider where each part of the partial intrinsic behavioral description for CONVT (see the figure above) comes from. The SUBROUTINE statement, line 31, specifies that there are six inputs and outputs. Names for them in the behavioral description have been patterned after the variable names for clarity. The system itself attaches no significance to its internal names.

This brings up the very interesting topic of mnemonic names. The variable names are a very important source of information about a program, particularly if there is not much other annotation. Since N, M, and MS are standard names for NUMBER_OF_ROWS, NUMBER_OF_COLUMNS, and STORAGE_MODE respectively, in the programs in the SSP, additional commentation on this fact might have been omitted if CONVT was not so well documented.

The name of a variable is usually chosen to specifiy some aspect of the datum carried by the variable. This can be used to learn about the assertions of the segment producing the datum, and about the prerequisites of segments using it. However, there is a problem with this. A variable is usually used to carry different datums, in different parts of the program, while the mnemonic name often applies to only some of these datums. For example, the variable NM in CONVT is very mnemonic when it is carrying the result of line 36, but not when it is carrying the result of line 38 or 40.

In addition to this problem, since the system does not understand English, it cannot use the mnemonic value of a name directly. As a result, the system will probably not use any more information from the variable names than FORTRAN uses, namely, the fact that the first letter of a variable name signifies whether it is an integer or floating point number.

Returning to the header comment, the DIMENSION statement, line 32, indicates that S and D are matrices. This fact is specifically recorded in the partial behavioral description. In addition S and D are referred to as {s(i)} and {d(i)} when talking about the entire aggregate. The prerequisites and assertions about the data types of the inputs and outputs come from the names for the variables and the DOUBLE PRECISION statement, line 33.

The specific information about N, M, MS, and MODE is given in lines 5-9 and 18. The information about what S and D are, on output, is given, though not too clearly, in lines 10-17 and 27-29. One source of confusion is that, unlike the partial behavioral description, the comment does not make a clear distinction between input and output data carried by the same variable. As a result, when it says "S," it is not clear whether it is the input or the output data which is being referred to.

The comments on lines 24-25 are not used because they do not apply to the internal workings of the program CONVT. The very important comment on line 23 is not used because this system makes the simplifying assumption that variables never overlap.

The partial intrinsic behavioral description derived from the header comment is used as a basis for the intrinsic behavioral description of the program. In fact, it is almost complete. The only thing that has to be added is some information about some of the outputs. Many comments are in the form of partial behavioral descriptions, but few are anywhere near as complete as a header comment is.

At this point, the system has a good idea of the highest and lowest level descriptions of the operation of the program. In order to understand the program more fully, it must link these two descriptions up by determining the intermediate structure of the program.

Basically, it is not too difficult for the system to analyze one or maybe two levels beyond what it knows. An attempt to go farther than that, would lead to a combinatorial explosion of possibilities. Internal comments in the program being analyzed provide landmarks so that the system never is more that two levels away from what it knows.

III.4.5 SEGMENTATION

The first step in developing an understanding of the internal workings of a program is to properly segment it. The basic difficulty involved in determining segmentation is that there is a very large number of ways that a program can be divided into segments. Fortunately, there are several methods which can be used to speed up the search.

Firstly there are comments which specify segmentation. The entire program is marked as a segment by being physically separated from the other programs in the SSP. Further, it is delineated by the SUBROUTINE and END statements, lines 31 and 52, for the benefit of the FORTRAN compiler.

The position of internal comments tends to specify internal segmentation though not unambiguously. The position of the comments in CONVT indicates that lines 35-40, 44-46, and 48-49 are segments. It is harder to see that line 41 indicates a segment from 42-49, not just for 42. The comment on line 50 is particularly interesting in that it is clearly intended solely to specify segmentation. In the last figure, segmentation information is indicated by indentation. This obviously would not be adequate in a situation where there were overlapping segments. It does suffice here, however.

Secondly, the segmentation can proceed incrementally (for one or two levels) from above or below. This is done in conjunction with recognizing plan types expressed in the code (see the next section). If a plan can be recognized, this automatically gives some of the segmentation. The reason this process can only proceed profitably from the top and bottom, and not from the middle, is that in order to recognize the plan, either the segment, or the subsegments must be determined.

In CONVT proceeding from the bottom can easily detect that the expressions are segments. Working from the top, it would not be too hard to notice that the whole program is a COMP of lines 35-40 and 42-49, and that these two segments are XORs of (35, 36, 38, and 40) and (42, 44-45, and 48-49) respectively. In this simple example the incremental approach yields a total analysis because the program is very shallow.

III.4.6 PLANS

The main reason recognition of the plan type of a segment is possible is that there are only a few plan types used in the SSP. There are an abundance of features which are very specific in differentiating between these plan types. For instance, all LOOPs, and only LOOPs, have a loop in control and data flow. All, and only, LOOPs and XORs have predicates. All, and only, COMPs and LOOPs have data flow between subsegemnts. In addition, the subcategories of the four major plan types also have clear identifying characteristics.

Of considerable aid to recognition is the fact that certain FORTRAN constructs are stereotypically used to implement certain plan types. For instance, ARITHMETIC IFs often implement XORs. DOs often implement AUGMENTED LOOPs, a DO being an ENUMERATION LOOP. Expressions implement sequences of COMPs.

Finally, comments sometimes give an indication of the plan type of a segment. The phrases "FIND STORAGE MODE" in line 34 and "TEST TYPE OF CONVERSION" in line 41 seem to indicate that the segments following them are XORs. In general, however, comments are not very helpful for discovering the plan type of a segment.

III.4.7 BEHAVIORAL DESCRIPTIONS

The internal comments are very useful for developing the behavioral descriptions of the internal segments. In CONVT, the comment on line 34 indicates that the output of the following segment is the number of elements in the arrays. The comment on line 43 indicates that the output is the DBLE of the input. Similarly, the comment on line 47 indicates that the output is the SNGL of the input.

With some vital guidance from the internal comments, information about behavioral descriptions for the internal segments comes up from the intrinsic behavioral descriptions of the primitive programs used, and down from the partial intrinsic behavioral description for the whole program given

by the header comment. Each plan type has information about what the intrinsic description of a segment corresponding to that plan type must be if the extrinsic behavioral descriptions of the subsegments are known. In the absence of a comment to the contrary, the extrinsic description of a segment is assumed to be the same as the intrinsic description. Information cannot propagate from the intrinsic behavioral description of a segment to the extrinsic behavioral descriptions of its subsegments unless something is known about the descriptions of these subsegments. Comments are crucial in providing enough information about the behavioral description of a subsegment so that a more complete description can be inferred from the description of the segment. This allows the high level description provided by the header comment to be pushed down into the program.

Information filters up from the primitive programs until it meets information filtering down from the header comment. It is at the places where those two types of information meet, that the system is required to perform significant deductions. These often take the form of proving that an extrinsic behavioral description, partially specified by a comment, follows from an intrinsic behavioral description, specified from below.

III.4.8 THE GRAND PLAN FOR CONVT

The previous three sections tried to give an idea of how an understanding of a program is developed. This section exhibits a complete grand plan for CONVT, and a discussion of how this specific grand plan can be developed. The next figure is a schematic of the grand plan for CONVT.

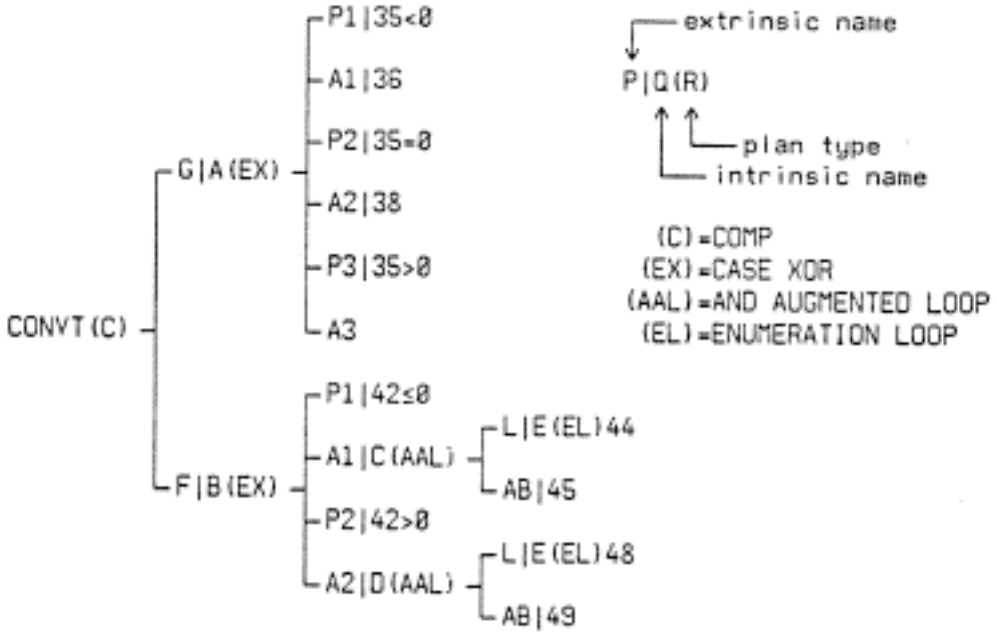


Fig. 29: Schematic of the grand plan for CONVT. This is a shorthand notation for a diagram of the grand plan similar to the diagrams for the plan types in section III.3. This figure shows how the plans are imbedded. The diagrams in section III.3 show what the diagram corresponding to each node of the schematic looks like.

The figure shows how the segments combine to produce the program CONVT. The next figure shows much of the same information in a different form. In addition, it clearly shows how the segments relate to the physical code.

```
1-30 are comments applying to the whole program.
31  V                       SUBROUTINE CONVT (N,M,MODE,S,D,MS)
32  V                       DIMENSION S(1),D(1)
33  V                       DOUBLE PRECISION D
34  V   (G)       C           FIND STORAGE MODE OF MATRIX AND NUMBER OF DATA POINTS
35  VA  (P1,P2,P3)          IF (MS-1) 2, 4, 6
36  VA  (A1)              2 NM=N*M
37  VA                      GO TO 8
38  VA  (A2)              4 NM=((N+1)*N)/2
39  VA                      GO TO 8
40  VA  (A3)              6 NM=N
41  V   (F)       C           TEST TYPE OF CONVERSION
42  VB  (P1,P2)           8 IF (MODE-1) 10, 10, 20
43  VB  (A1)      C           SINGLE PRECISION TO DOUBLE PRECISION
44  VBCE                 10 DO 15 L=1,NM
45  VBC (AB)             15 D(L)=S(L)
46  VB                      GO TO 30
47  VB  (A2)      C           DOUBLE PRECISION TO SINGLE PRECISION
48  VBDE                 20 DO 25 L=1,NM
49  VBD (AB)             25 S(L)=D(L)
50  V   (F)       C
51  V                    30 RETURN
52  V                       END
```

Fig. 30: The program CONVT showing segmentation information. Each line is preceded by the intrinsic name of every segment containing it (V=CONVT). In addition, in parenthesis, are the extrinsic name(s) of the segment(s) the line is most closely associated with (if appropriate).

Both figures stop at the level of expression oriented basic segments which can easily be built up from the actual primitive programs. As discussed in the section on segmentation (III.4.5) and on plans (III.4.6), a multiplicity of factors allows the segmentation and plan types to be inferred. Most notably, the DO statements in lines 44 and 48 indicate the extents of segments C and D, and that they are AUGMENTED LOOPs. The ARITHMETIC IF statements in lines 35 and 42 together with the dividing and then rejoining character of the control flow diagram indicates the extent of segments A and B and that they are CASE XORs. The appearance of the control and data flow diagrams indicates that the segment CONVT is a COMP.

Given the segmentation structure, the system then fleshes out the grand plan with behavioral descriptions. The figure at the end of this section shows the complete grand plan for CONVT. The grand plan is large. However, it is straightforwardly derivable from the text for CONVT.

All of the data and control flow information is derived from the data and control flow diagrams. The intrinsic descriptions of the basic units (which are at the end of the figure) are developed from the specific intrinsic behavioral descriptions of the primitive elements known by the system and used in CONVT. Lower levels of detail have been suppressed for simplicity.

Consider the following scenario for how the other behavioral descriptions were developed. Starting from the top, the basic intrinsic behavioral description for CONVT is taken from the header comment. A few clauses about the outputs have filtered up from below. Otherwise, it is the same as in the comment translation figure. The comments on lines 34, 41, 43, and 47 make it possible to propagate some of the high level information in the intrinsic behavioral description for CONVT into segment B, and to flesh out the plan for the segments CONVT and B. They do this by indicating key parts of the extrinsic behavioral descriptions of A, B, C, and D.

Working from the bottom, there is no difficulty in developing the complete plan for segment A. Similarly there is no difficulty in developing the plans for the AND AUGMENTED LOOPs C and D. A plan for the ENUMERATION LOOP E is included in the figure so that it can be seen how parts of it are used

to develop the plans for C and D. (refer to the section on AUGMENTED LOOPs III.3.4.3).

It should be noticed, that the intrinsic description of the ENUMERATION LOOP segment E differs considerably from the extrinsic behavioral descriptions of its use in segements C and D. This difference embodies a theorem which is probably too difficult for the system to deduce. The fact that the programmer chose to use a DO statement to implement E causes the system to use the indicated extrinsic behavioral description. The intrinsic description of E is one that would arise if the programmer had not used a DO but rather had open coded the LOOP. It is included to give added insight into the plans for LOOPs.

As discussed in section III.1.3.6, the array indexing in lines 45 and 49 has been factored out of the computation into the data flow. This can be done because, as the plan for segement E shows, the value of the variable L is a function of the number of times the loop has been executed, not of any data value. This system makes this type of factorization whenever possible. As in this case, it usually leads to a significant simplification of the statement of the plans.

As an example of the process of developing behavioral descriptions, consider segment D. The DO on line 48 indicates that lines 48 and 49 form a segment, and that that segment is an AUGMENTED LOOP. DO is a primitive construct, and so the system immediately produces an intrinsic behavioral description for it (see the intrinsic behavioral description of segment E in the figure at the end of this section). Referring to section III.3.4.1, it can be seen that this description conforms to the general pattern of a description of a LOOP, as expressed by the equations in that section. Segment E is an ENUMERATION LOOP since it has no outputs. The assertions of E are just a combination of the assertions of its body and test.

Because segment E is implemented by a DO, the system knows that there is an equivalent and much more useful way to state its assertions. This is used in the extrinsic description of segment E when used in segment D (see the figure at the end of this section). Line 49 can be identified as the AB (additional body) of the AUGMENTED LOOP. Section III.3.4.3 shows how an AB is integrated into a LOOP in order to form an AUGMENTED LOOP. In this case, the data flow diagram shows that the AB (line 49) is executed before the body of the original LOOP and takes the output of the previous execution of that body (the quantity named k[i] in the description of E) as an input. This quantity is carried by the variable L. The assertions of E (namely that k[i]=i+1) show that the value of L depends only on the number of times the LOOP has cycled. As a result, the array references in line 49 can be factored out of the computation into the data flow. This makes it clear that AB (line 49) does not feed back to itself because the {s(i)} are not actually input to it. This implies that D is an AND AUGMENTED LOOP.

Referring again to section III.3.4.3, the intrinsic behavioral description of D can easily be inferred. The assertions of D are just a combination of the assertions of AB and segment E. The assertions of E have actually been omitted except for the fact that the LOOP is executed from 0 to limit (since k[i] is not an output of D). Note that the only deduction required in this whole process was pattern matching and that there were not a large number of blind alleys which the system had to follow.

The teleological links (see section III.2.2) are omitted from the grand plan due to the lack of a reasonable way to represent them. The key deductions take place when the information propagated up from below first meets information filtered down from above. This happens at the interface between the intrinsic and extrinsic behavioral descriptions of the segments A, C, and D. In order to demonstrate consistency at these points the system must use theorems defining the terms "NUMBER_OF_ELEMENTS" and "= COMPONENTWISE".

given:

matrix ({x(i)})
n=NUMBER_OF_ROWS({x(i)})
m=NUMBER_OF_COLUMNS({x(i)})
ms=STORAGE_MODE({x(i)})
ne=NUMBER_OF_ELEMENTS({x(i)})

then:

n>0
m>0
ms ∈ {1, 2, 3}
ms=0 → ne=n*m
ms=1 → ne=((n+1)*n)/2
ms=2 → ne=n
ne>0

given:

matrices ({x(i)}, {y(i)})
ne=NUMBER_OF_ELEMENTS({x(i)})=NUMBER_OF_ELEMENTS({y(i)})

then:

{x(i)} = F({y(i)}) COMPONENTWISE ≡ $\wedge_{i=1,ne}$ x(i) = F(y(i))

Fig. 31: Theorems about matrices needed for understanding of CONVT

The first theorem shows that segment A does indeed calculate the NUMBER_OF_ELEMENTS in the matrices. The second theorem shows that segments C and D do indeed produce outputs which are COMPONENTWISE functions of their inputs. These deductions close the gaps between the intrinsic and extrinsic behavioral descriptions of these three segments.

It is reasonable to expect that the system might know these theorems as part of its knowledge of matrices, which are the only complex data type in FORTRAN. If not, then the user would either have to give them as part of the header comment or the system would ask him if the deductions required above were valid. If he said they were, then the system would in effect asume these theorems to be true, though it would not use them in any other context.

If anywhere along the line the system discovered any inconsistency in the program, it would report a bug as discussed in section II.2. The program CONVT does not have any bugs. See section II.2 for a discussion of the program RK1 which does have bugs.

Finally, in a tidying up phase, the system propagates low level information all the way up to the top, and fills in gaps in the intrinsic behavioral description for the whole program. In order not to keep a lot of excess information, it only adds in information to fill in conspicuous gaps. In this case, several of the outputs to the program (such as on, om, and {os(i)} when mode=1) are completely unspecified. Data flow analysis shows that they are directly mapped from inputs.

III.4.9 TRANSFORMATIONS

The only transformation (see section III.1.3) which was applied to produce the program CONVT, is factoring computation out of the flow of control. An arithmetic IF was used to implement several predicates at once in line 35 and line 42. This is easy for the system to spot by looking at the control flow diagram. The system undoes the transformation before analyzing the program further. Other transformations can also be identified by clues indicating that they have been applied.

It should be stressed that transformations are a thorny issue. It is essential that the system be very conservative about applying inverse transformations to a program. If the system experiments with reversing a large number of transformations which might have applied at any given point, it will drown in a sea of alternative programs. If a large number of transformations have been applied to a

program, comments will undoubtedly be needed so that the program can be unscrambled and understood.

Further, it should be noted, that though a transformation may have been applied to a program while it was being written, the inverse transformation may not have to be applied in order to understand the program. Consider the following two programs which are related to each other by factoring.

```
   IF (I) 10,10,20
10 Y=SIN(X)
   W=X*X
   GOTO 30
20 Y=SIN(Z)
   W=X*X
30
```

```
   IF (I) 10,10,20
10 Y=SIN(X)
   GOTO 30
20 Y=SIN(Z)
30 W=X*X
```

Fig. 32: Two programs related by factoring.

The second program can be understood as an AND of an XOR and "W=X*X", rather that as a transformed XOR of two ANDs. Only transformations leading to distorted programs not fitting any plan type need be undone. In the second program, a comment might well be inserted to indicate that there is a transformation which should be undone. Otherwise, there is no reason to think that a transformation has applied.

Fig. 33: The grand plan for CONVT

```
Plan for segment CONVT which is a COMP of G|A and F|B
        extrinsic description of G|A
                inputs: ms, n, m
                prerequisites: integer numbers (ms, n, m)
                        matrices ({s(i)}, {d(i)})
                        n=NUMBER_OF_ROWS({s(i)})=NUMBER_OF_ROWS({d(i)})
                        m=NUMBER_OF_COLUMNS({s(i)})=NUMBER_OF_COLUMNS({d(i)})
                        ms=STORAGE_MODE({s(i)})=STORAGE_MODE({d(i)})
                outputs: length
                assertions: integer number (length)
                        length=NUMBER_OF_ELEMENTS({s(i)})=NUMBER_OF_ELEMENTS({d(i)})
                mapping: (ms↔type, n↔n, m↔m, length↔size;)
        extrinsic description of F|B
                inputs: {d(i)}, {s(i)}, length, mode
                prerequisites: floating numbers ({s(i)})
                        double precision numbers ({d(i)})
                        matrices ({s(i)}, {d(i)})
                        integer numbers (length, mode)
                        length=NUMBER_OF_ELEMENTS({s(i)})=NUMBER_OF_ELEMENTS({d(i)})
                        mode ∈ {1,2}
                outputs: {od(i)},{os(i)}
                assertions: floating numbers ({os(i)})
                        double precision numbers ({od(i)})
                        matrices({os(i)}, {od(i)})
                        mode=1 → {od(i)}=DBLE({s(i)}) COMPONENTWISE
                                        ∧ {os(i)}={s(i)} COMPONENTWISE
                        mode=2 → {os(i)}=SNGL({d(i)}) COMPONENTWISE
                                        ∧ {od(i)}={d(i)} COMPONENTWISE
                mapping: ({d(i)}↔{d(i)}, {s(i)}↔{s(i)}, length↔length, mode↔mode,
                        {os(i)}↔{os(i)}, {od(i)}↔{od(i)};)
        intrinsic description of CONVT
                inputs: n, m, mode, {s(i)}, {d(i)}, ms
                prerequisites: floating numbers ({s(i)})
                        double precision numbers ({d(i)})
                        integer numbers (n, m, mode, ms)
                        matrices ({s(i)}, {d(i)})
                        n=NUMBER_OF_ROWS({s(i)})=NUMBER_OF_ROWS({d(i)})
                        m=NUMBER_OF_COLUMNS({s(i)})=NUMBER_OF_COLUMNS({d(i)})
                        ms=STORAGE_MODE({s(i)})=STORAGE_MODE({d(i)})
                        mode ∈ {1, 2}
                outputs: on, om, omode, {os(i)}, {od(i)}, oms
                assertions: floating numbers ({os(i)})
                        double precision numbers ({od(i)})
                        integer numbers (on, om, omode, oms)
                        matrices({os(i)}, {od(i)})
                        mode=1 → {od(i)}=DBLE({s(i)}) COMPONENTWISE
                                        ∧ {os(i)}={s(i)} COMPONENTWISE
                        mode=2 → {os(i)}=SNGL({d(i)}) COMPONENTWISE
                                        ∧ {od(i)}={d(i)} COMPONENTWISE
                        on=n
                        om=m
                        omode=mode
```

```
                    oms=ms
data flow
        into G|A
                ms from outside CONVT via argument position 6
                        and variable MS
                n from outside CONVT via argument position 1
                        and variable N
                m from outside of CONVT via argument position 2
                        and variable M
        into F|B
                length from length of G|A via variable NM
                mode from outside of CONVT via argument position 3
                        and variable MODE
                {s(i)} from outside of CONVT via argument position 4
                        and variable S
                {d(i)} from outside of CONVT via argument position 5
                        and variable D
        to outside of CONVT
                on from outside of CONVT via argument position 1,
                        variable N, and argument position 1
                om from outside of CONVT via argument position 2,
                        variable M, and argument position 2
                omode from outside of CONVT via argument position 3,
                        variable MODE, and argument position 3
                oms from outside of CONVT via argument position 6,
                        variable MS, and argument position 6
                {os(i)} from {os(i)} of F|B via variable S
                        and argument position 4
                {od(i)} from {od(i)} of F|B via variable D
                        and argument position 5
control flow
        from outside of CONVT to G|A via initial placement
        from G|A to F|B via sequential placement
        from F|B to outside of CONVT by sequential placement
                and RETURN line 51
```

```
Plan for segment A which is a CASE XOR of P1|35<0, A1|36, P2|35=0, A2|38, P3|35<0, and A3
        extrinsic description of P1|35<0
        (the IF line 35 predicates the segment 35 taking the + branch when result<0)
                inputs: type
                prerequisites: integer number (type)
                outputs: none
                assertions: type<1
                mapping: (type↔arg1, result↔return_value;)
        extrinsic description of A1|36
                inputs: n, m
                prerequisites: integer numbers (n, m)
                outputs: size
                assertions: integer number (size)
                        size=n*m
                mapping: (n↔arg1, m↔arg2, size↔return_value;)
        extrinsic description of P2|35=0
        (the IF line 35 predicates the segment 35 taking the + branch when result=0)
                inputs: type
                prerequisites: integer number (type)
                outputs: none
                assertions: type=1
                mapping: (type↔arg1, result↔return_value;)
        extrinsic description of A2|38
                inputs: n
                prerequisites: integer number (n)
                outputs: size
                assertions: integer number (size)
                        size=((n+1)*n)/2
                mapping: (n↔arg1, n↔arg2, size↔return_value;)
        extrinsic description of P3|35>0
        (the IF line 35 predicates the segment 35 taking the + branch when result >0)
                inputs: type
                prerequisites: integer number (type)
                outputs: none
                assertions: type>1
                mapping: (type↔arg1;)
        extrinsic description of A3
                inputs: n
                prerequisites: integer number (n)
                outputs: size
                assertions: integer number (size)
                        size=n
                mapping: (;size↔n)
        intrinsic description of A
                inputs: type, n, m
                prerequisites: integer number (type)
                        type<1 → integer numbers (n, m)
                        type=1 → integer number (n)
                        type>1 → integer number (n)
                outputs: size
                assertions: integer number (size)
                        type<1 → size=n*m
                        type=1 → size=((n+1)*n)/2
                        type>1 → size=n
```

```
data flow
        into P1|35<0
                type from outside A via variable MS
        into A1|36
                n from outside A via variable N
                m from outside A via variable M
        into P2|35=0
                type from outside A via variable MS
        into A2|38
                n from outside A via variable N
        into P3|35>0
                type from outside A via variable MS
        into A3
                n from outside A via variable N
        to outside of A
                size from size of A1|36 via variable NM
                        or from size of A2|38 via variable NM
                        or from size of A3 via variable NM
control flow
        from outside of A to P1 via initial placement
        from P1 to A1 via IF line 35 and label 2
        from P1 to P2 via shared code line 35
        from P2 to A2 via IF line 35 and label 4
        from P2 to P3 via shared code line 35
        from P3 to A3 via IF line 35 and label 6
        from A1 to outside of A via GOTO line 37 and label 8
        from A2 to outside of A via GOTO line 39 and label 8
        from A3 to outside of A via final placement
```

Plan for segment B which is a CASE XOR of P1|42≤0, A1|C, P2|42>0, and A2|D
extrinsic description of P1|42≤0
(the IF line 42 predicates the segment 42 taking the +branch when result≤0)
inputs: mode
prerequisites: integer number (mode)
mode ∈ {1,2}
outputs: none
assertions: mode=1
mapping: (mode↔arg1, result↔return_value;)
extrinsic description of A1|C
inputs: length, {s(i)}
prerequisites: floating numbers ({s(i)})
integer number (length)
length=NUMBER_OF_ELEMENTS({s(i)})=NUMBER_OF_ELEMENTS({d(i)})
outputs: {os(i)}, {od(i)}
assertions: floating numbers ({os(i)})
double precision numbers ({od(i)})
matrices({os(i)}, {od(i)})
{od(i)}=DBLE({s(i)}) COMPONENTWISE
∧ {os(i)}={s(i)} COMPONENTWISE
mapping: (length↔limit, {s(i)}↔{source(i)}, {od(i)}↔{dest(i)}; {os(i)}↔{s(i)})
extrinsic description of P2|42>0
(the IF line 42 predicates the segment 42 taking the + branch when result>0
inputs: mode
prerequisites: integer number (mode)
mode ∈ {1,2}
outputs: none
assertions: mode=2
mapping: (mode↔arg1, result↔return_value;)
extrinsic description of A2|D
inputs: length, {d(i)}
prerequisites: double precision numbers ({d(i)})
integer number (length)
length=NUMBER_OF_ELEMENTS({d(i)})=NUMBER_OF_ELEMENTS({s(i)}
outputs: {os(i)}, {od(i)}
assertions: floating numbers ({os(i)})
double precision numbers ({od(i)})
matrices({os(i)}, {od(i)})
{os(i)}=SNGL({d(i)}) COMPONENTWISE
∧ {od(i)}={d(i)} COMPONENTWISE
mapping: (length↔limit, {d(i)}↔{source(i)}, {os(i)}↔{dest(i)}; {od(i)}↔{d(i)})
intrinsic description of B
inputs: mode, length, {s(i)}, {d(i)}
prerequisites: integer numbers (mode, length)
length=NUMBER_OF_ELEMENTS({s(i)})=NUMBER_OF_ELEMENTS({d(i)})
mode ∈ {1,2}
mode=1 → floating numbers ({s(i)})
mode=2 → double precision numbers ({d(i)})
outputs: {os(i)}, {od(i)}
assertions: floating numbers ({os(i)})
double precision numbers ({od(i)})
matrices({os(i)}, {od(i)})
mode=1 → {od(i)}=DBLE({s(i)}) COMPONENTWISE
∧ {os(i)}={s(i)} COMPONENTWISE

```
                    mode=2 → {os(i)}=SNGL({d(i)}) COMPONENTWISE
                                        ∧ {od(i)}={d(i)} COMPONENTWISE
data flow
        into P1|42≤0
                switch from outside B via variable MODE
        into A1|C
                {s(i)} from outside B via variable S
        into P1|42>0
                switch from outside B via variabel MODE
        into A2|D
                {d(i)} from outside B via variable D
        to outside of B
                {os(i)} from {os(i)} of A1 via variable S
                        or from {os(i)} of A2 via variable S
                {od(i)} from {od(i)} of A1 via variable D
                        or from {od(i)} of A2 via variable D
Control flow
        from outside of B to P1|42≤0 via initial placement
        from P1|42≤0 to A1|C via If line 42 and label 10
        from P1|42≤0 to P2|42>0 via shared code line 42
        from P2|42>0 to A2|D via IF line 42 and label 20
        from A1 to outside of B via GOTO line 46 and label 30
        from A2 to outside of B via final placement
```

```
Plan for segment C which is an AND AUGMENTED LOOP of L|E and AB|45
        extrinsic description of L|E
                inputs: limit
                prerequisites: integer number (limit)
                        limit>0
                outputs: none
                assertions: integer numbers ({k[i]})
                        ^i=0,limit k[i]=i+1
                mapping: (limit↔endvalue, k↔k;)
        extrinsic description of AB[0] (part of I)
                inputs: none
                prerequisites: none
                outputs: none
                assertions: none
                mapping: (;)
        extrinsic description of AB[i]|45 i=1,∞
                inputs: source(i)
                prerequisites: floating number (source(i))
                outputs: dest(i)
                assertions: double precision number (dest(i))
                        dest(i)=DBLE(source(i))
                mapping: (source(i)↔fnum, dest(i)↔dnum;)
        intrinsic description of C
                inputs: limit,{source(i)}
                prerequisites: floating numbers ({source(i)})
                        integer number (limit)
                        limit>0
                outputs: {dest(i)}
                assertions: double precision numbers ({dest(i)})
                        ^i=1,limit(dest(i)=DBLE(source(i))
        data flow
                into L|E
                        limit from outside C via variable NM
                into AB[i]|45
                        source(i) from ({source(i)} which comes from outside of C
                                via variable S) via variable S(L)
                to outside of C
                        {dest(i)} from (the dest(i) which come from dest(i)
                                of {AB[i]|45} via variable D(L)) via variable D
        control flow
                from outside of C to I (L.I and AB[0]) via initial placement
                        I initiates the LOOP between T and AB
                from T to AB via DO line 44
                from AB to L.B via DO line 44
                from L.B to T via DO line 44
                from T to outside of C via DO line 44
```

Plan for segment D which is an AND AUGMENTED LOOP of L|E and AB|49
- extrinsic description of L|E
  - inputs: limit
  - prerequisites: integer number (limit)
    - limit>0
  - outputs: none
  - assertions: integer numbers ({k[i]})
    - $\wedge_{i=0,limit} k[i]=i+1$
  - mapping: (limit↔endvalue, k↔k$_i$)
- extrinsic description of AB[0] (part of I)
  - inputs: none
  - prerequisites: none
  - outputs: none
  - assertions: none
  - mapping: (;)
- extrinsic description of AB[i]|49 i=1,∞
  - inputs: source(i)
  - prerequisites: double precision number (source(i))
  - outputs: dest(i)
  - assertions: floating number (dest(i))
    - dest(i)=SNGL(source(i))
  - mapping: (source(i)↔dnum, dest(i)↔fnum$_i$)
- intrinsic description of C
  - inputs: limit,{source(i)}
  - prerequisites: double precision numbers ({source(i)})
    - integer number (limit)
    - limit>0
  - outputs: {dest(i)}
  - assertions: floating numbers ({dest(i)})
    - $\wedge_{i=1,limit}$(dest(i)=SNGL(source(i))
- data flow
  - into L|E
    - limit from outside D via variable NM
  - into AB[i]|49
    - source(i) from ({source(i)} which comes from outside D via variable D) via variable D(L)
  - to outside of D
    - dest(i) from (the dest(i) which come from dest(i) of {AB[i]|49} via variable S(L)) via variable S
- control flow
  - from outside of C to I (L.I and AB[0]) via initial placement
    - I initiates the LOOP between T and AB
  - from T to AB via DO line 48
  - from AB to L.B via DO line 48
  - from L.B to T via DO line 48
  - from T to outside of D via DO line 48

Plan for segment E which is an ENUMERATION LOOP of I, B, and T
(literally realized in a DO line 44 and in a DO line 48)
(mappings, data and control flow are omitted since the DO statement is self contained)

extrinsic description of B[0] (part of I)
- inputs: none
- prerequisites: none
- outputs: k[0]
- assertions: integer number (k[0])
  - k[0]=1

extrinsic description of B[i] i=1,∞
- inputs: k[i-1]
- prerequisites: integer number (k[i-1])
- outputs: k[i]
- assertions: integer number (k[i])
  - k[i]=k[i-1]+1

extrinsic description of T[0] (part of I)
(this predicate is an inherent part of a DO statement)
- inputs: none
- prerequisites: none
- outputs: none
- assertions: none

extrinsic description of T[i] i=1,∞
(this predicate is an inherent part of a DO statement)
- inputs: k[i], endvalue
- prerequisites: integer numbers (k[i], endvalue)
- outputs: none
- assertions: k[i]≤endvalue

intrinsic description of E
- inputs: endvalue
- prerequisites: integer number (endvalue)
- outputs: none
- assertions: integer numbers ({k[i]})
  - k[0]=1
  - $\wedge_{i=1,n} k[i]=k[i-1]+1$
  - $\wedge_{i=1,n-1} k[i]\leq endvalue$
  - k[n]>endvalue

Intrinsic descriptions of the basic segments: 35, 36, 38, 42, 45, and 49

intrinsic description of 35
intrinsic description of 35
inputs: arg1
prerequisites: integer number (arg1)

outputs: return_value
assertions: integer number (return_value)
return_value=arg1-1

intrinsic description of 36
inputs: arg1, arg2
prerequisites: integer numbers (arg1, arg2)
prerequisites: integer numbers (arg1, arg2)
outputs: return_value
assertions: integer number (return_value)
return_value=arg1*arg2

intrinsic description of 38
inputs: arg1, arg2
prerequisites: integer numbers (arg1, arg2)
outputs: return_value
assertions: integer number (return_value)
return_value=((arg1+1)*arg2)/2

intrinsic description of 42
inputs: arg1
prerequisites: integer number (arg1)
outputs: return_value
assertions: integer number (return_value)
return_value=arg1-1

intrinsic description of 45
inputs: fnum
prerequisites: floating number (fnum)
outputs: dnum
assertions: double precision number (dnum)
dnum=DBLE(fnum)

intrinsic description of 49
inputs: dnum
prerequisites: double precision number (dnum)
outputs: fnum
assertions: floating number (fnum)
fnum=SNGL(dnum)

Bauer, M. (1975) "A basis for the Acquisition of Procedures from Protocols", Fourth International Joint Conference on A.I. U.S.S.R.

Boyer, Robert and Moore, Strother (1975) "Proving Theorems About LISP Programs", JACM V22 #1 Jan. 1975, pp. 129-244

Brown, A.L. (1974) "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures" MIT AI-WP-61 March 1975

Brown, A.L. (1975) "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures" MIT PhD thesis Sept. 1975

Brown, A.L. (forthcoming) "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures" MIT AI-TR-362

Floyd, R.W. (1967) "Assigning Meaning to Programs", Proc. Symposia in applied math. V19 Am. Math. Soc. pp. 19-32 Prov. R.I.

Floyd, R.W. (1971) "Toward Interactive Design of Correct Programs" Stanford AIM 158 Sept 1971

Gerhart, S.L. (1975) "Knowledge About Programs; a Model and a Case Study", SIGPLAN Notices, V10 #6 Proc of International Conf. on Reliable Software June 1975

Goldstein, Ira (1974) "Understanding Simple Picture Programs" PhD thesis M.I.T. MIT-AI-TR-294

Goldstein, I. (1976) "Planning Paradigms - Knowledge for Organizing Models into Programs" MIT AI-WP-123 March 1976

Green, C.C. et. al. (1974) "Progress Report on Program Understanding Sytems" Stanford AIM-240 August 1974

Green, C.C. and Barstow, D. (1975) "A Hypothetical Dialogue Exhibiting a Knowledge Base for a Program Understanding System" Stanford AIM-258 (STAN-CS-75-476) Jan. 1975

Hammer, M., Howe, W. G., Kruskal, V. J., and Wladawsky I. (1975) "A Very High Level Programming Language for Data Processing Applications" IBM research report RC-5583 Yorktown Heights N.Y.

Hewitt, C. and Smith, B. (1975) "Towards a Programming Apprentice" IEEE Transactions on Software Engineering Vse-1 #1 pp. 26-46 March 1975

Hardy, S. (1975) "Synthesis of LISP Functions from Examples", Fourth Internation Joint Conference on A.I. U.S.S.R.

Hoare, C.A.R. (1969) "An Axiomatic Basis for Computer Programming", CACM V12 #10 pp. 576-583

Hoare, C.A.R. (1971) "Procedures and Parameters: An Axiomatic Approach", Symposia on the Semantics of Algorithmic Languages E. Engeler ed. pp. 102-116, Springer

IBM GH20-0205-4 (1970) "Scientific Subroutine Package Version III Programmer's Manual" White Plains N.Y.

Liskov, B. (1974) "A Note on CLU" M.I.T. Computation Structures Group Memo 112

Malhotra, A. and Sheridan P.B. (1976) "Experimental Determination of Design Requirements for a Program Understanding System" IBM research report RC-5831 Jan 1976 Yorktown Heights N.Y.

Manna, Z. and Waldinger, R. (1975), "Knowledge and Reasoning in Program Synthesis", Artificial Intellegence V6 pp. 175-208

Mikelsons, M. and Wladawski I. (1976) "On the Formal Documentation of Programs" IBM research report 1976 Yorktown Heights N.Y.

Moore, J.S. (1974) "Introducing PROG into the Pure LISP Theorem Prover", Xerox PARC report CSL-74-3

Rich, C. and Shrobe, H.E. (1974) "Understanding LISP Programs: Towards a Programmers Apprentice" MIT AI-WP-82 Dec. 1974

Rich, C. and Shrobe, H.E. (1976) "Initial report on a LISP Programmers Apprentice" NIT Masters Thesis, AI-TR-354

Ruth, G.R. (1974) "Analysis of Algorithm Implementations" MIT PhD Thesis, MIT MAC-TR-130 May 1974

Ruth, G.R. (1976) "Protosytem I: an Automatic Programming System Prototype", M.I.T. LCS-TM-72

Shaw, D., Swartout, W., and Green C. (1975) "Inferring LISP Programs From Examples", Fourth INternational Joint Conference on A.I., U.S.S.R.

Smith, B., Waters, R.C., and Lieberman, H. (1973) "Comments on Comments or the Purpose of Intentions and the Intention of Purposes" Term project for MIT course 6.893 "Automating Knowledge Based Programming and Validation Using ACCTRS" Dec. 1973

Smith, B. and Hewitt C. (1974) "Towards a Programming Apprentice" AISB summer conference July 1974

Summers, P.D. (1975) "Program Construction from Examples", PhD thesis Yale Univ.

Sussman, G.J. (1973a) "A Computational Model of Skill Acquisition" PhD. thesis M.I.T. AI-TR-297 August 1973

Sussman, G.J. (1973b) "A Scenario of Planning and Debugging in Electronic Circuit Design" MIT AI-WP-54 Dec. 1973

Sussman, G.J. (1974) "The Virtuous Nature of Bugs" Proceedings of theAISB Summer conference July 1974 pp. 224-237 U. of Sussex England

Waldinger, R. and Levitt, K.N. (1974) "Reasoning About Programs", Artificial Intelligence V5, pp. 235-316

Wegbreit, B. (1973) "Heuristic Methods for Mechanically Deriving Inductive Assertions", Third INternational Joint Conference on A.I. Stanford Univ.

Winograd, T. (1973) "Breaking the Complexity Barrier (Again)", Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973

Yonesawa, A. (1976) "Symbolic Evaluation as an Aid to Program Synthesis" MIT AI-WP-124 April 1976