INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in “sectioning” the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from “photographs” if essential to the understanding of the dissertation. Silver prints of “photographs” may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
MUTSCHLER, III, Eugene Otto, 1947-
AN INTEGRATED MODEL FOR COMPUTATIONAL
PROCESSES.

Rice University, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

Copyright
Eugene Otto Mutschler, III
1973
RICE UNIVERSITY

AN INTEGRATED MODEL FOR COMPUTATIONAL PROCESSES

by

Eugene Mutschler, III

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Thesis Director's Signature

Edward Alvin Feustel

Houston, Texas

May, 1973
# TABLE OF CONTENTS

**ACKNOWLEDGMENTS**

I. INTRODUCTION ........................................ 1
   EXISTING MODELS AND SYSTEMS ......................... 6
      ALGOL 60 AND THE BACKUS-NAUR FORM ............ 6
      THE ISP NOTATION OF BELL AND NEWELL .......... 10
      THE CONTOUR MODEL .............................. 12
      THE VIENNA DEFINITION LANGUAGE ............... 19
      THE MULTICS SYSTEM ............................. 27

II. PRESENTATION OF THE MODEL ......................... 32
    A PROCESS MACHINE ................................ 34
    ELEMENTS OF THE FORMAL MODEL .................. 36
       OBJECTS ........................................ 36
       VARIABLES ..................................... 48
    THE PROCESS MACHINE ............................... 56
    THE DEFINITION OF THE PROCESSOR ............... 58
    DISCUSSION ....................................... 60

III. RESULTS AND EQUIVALENCES WITH OTHER MODELS .... 64
     ON THE INVOCATION OF THE MODEL ................ 64
     THE ISP NOTATION ............................... 65
        MEMORY EQUIVALENCE LEMMA .................. 69
        EQUIVALENCE RESULT .......................... 71
     THE CONTOUR MODEL ............................... 73
     DATA TYPES ...................................... 80
     THE EQUIVALENCE MACHINE ....................... 81
     AUXILIARY FUNCTIONS ............................ 85
     EQUIVALENCE LEMMA AND PROOF ................... 88
     THE VIENNA DEFINITION LANGUAGE ............... 93
     AN OUTLINE OF A PROOF OF EQUIVALENCE ........ 93
     PROBLEMS ....................................... 98
     ADVANTAGES OF I-MODEL NOTATION ............... 98
ACKNOWLEDGEMENTS

There are many people without whose aid and comfort this thesis could not have been completed. The first is of course Dr. E. A. Feustel who was a treasure trove of ideas and materials. Thanks are also due to Drs. S. K. Rusk and K. W. Kennedy for serving on the thesis committee. Rod Bohlmann, Mike Donegan, and the rest of the staff of the Laboratory for Computer Science and Engineering provided hours of illuminating discussions. Scott Warren was most helpful as devil's advocate. John Doerr was most generous in lending his drafting equipment for the figures.

Very special thanks are due to Mr. J. K. Iliffe who started the author out in 1969 and who would see that some of his ideas have taken root and grown.
CHAPTER I
INTRODUCTION

The stored program computer has a fairly distinctive internal organization, which has changed little in the course of its development. This organization should be familiar to anyone acquainted with computers. However, a system diagram of a typical computer is reproduced below (Figure I-1) for reference. It is not hard to trace the effect of advances in component technology on the nature of computers. Due to the vacuum tube and discrete component technology used in early computers, the central processing organ comprised the bulk of the instrument. Machines with minimal input-output and memory facilities were of necessity bulky, expensive, and consumed huge amounts of power. For example, Von Neumann's early computer at the Institute for Advanced Studies at Princeton (the IAS Computer) was constructed with 4500 vacuum tubes and weighed one half ton. It had only a few hundred words of memory and virtually no input-output facilities as are in common use at the present time [I.2]. It was in fact one of the smaller computers of its time.

Progress in component technology, leading to the development and wide usage of high speed integrated circuits, has caused a virtual revolution in computer hardware, though the basic structure is usually that of Figure I-1.
Figure I-1.
Typical Computer Configuration. (After Gear [I.1]).
Computers still occupy entire basements; these basements are now filled with a variety of peripheral units and communications facilities. Central processors occupy small boxes in the heart of the system, yet are orders of magnitude more sophisticated than early processors. For example, the instruction execution time for the Digital Equipment Corporation (DEC) PDP-11 is roughly one tenth that of the IAS Computer mentioned above. The logic for the central processor is mounted on ten circuit boards of approximately 50 square inches each. The processor and 8192 words of memory are encased in a single cabinet and may be easily carried by an adult. The PDP-11, which may be obtained for less than $10,000 may control peripheral equipment worth an order of magnitude more.

The PDP-11 is considered a "mini" computer [I.3], by which is meant that it is designed and built to be small and inexpensive, at the cost of being much less powerful than more expensive machines. The larger computers on the market include as features more powerful instruction sets and larger word sizes. They may present the user with a "virtual" machine with almost unlimited flexibility, power, and memory. The IBM 360/67, for example, offers no less than several hundred instructions, including 14 ADD orders and separate instruction sets for binary and decimal arithmetic, as well as an addressing capability of 256 million 32-bit words [I.4].
Progress comes at a price, however. Von Neumann of necessity had to hand code the routines for his IAS computer in machine language [1.5]. Having been the guiding force behind the design and construction of that particular computer, Von Neumann was in a position to know exactly how the machine would execute his programs. Memory size limited program size to a few hundred words, each carefully chosen to perform the maximum useful work in minimum space and time. He was able to perform a complete analysis of results, including significance and roundoff losses. It must be said that the problems whose solutions he sought had to be amenable to such analyses to be considered for computer solution, computer time being a valuable commodity [1.6]. It must be further said that the time he spent in performing his analyses represented a great expense in itself. The fact remains, however, that by the time a program was declared "working" the programmer was on intimate terms with each instruction and its contribution to the solution of the problem at hand.

One finds all sort of problem submitted to a computer for solution in the current era. These include applications vastly more complex than ever before. In contrast to Von Neumann and his contemporaries a problem solver of today may or may not know how the computer actually performs the calculations necessary for a solution to his problem. He usually does not even begin to care until the results he
obtains are found to be erroneous. His subtraction of two nearly equal quantities may destroy significance; spurious interactions with other programs in a time sharing system may cause random errors or complete program failure. Moreover,

It is the unprofessional user who is more likely to complain of the restrictive rules of the language he has to employ and of the 'moronic' behaviour[sic] of the machine when a situation arises for which he has not provided, and he might reasonably ask whether we need to perpetuate such a state of affairs [I.7].

System designs must be found to minimize these as well as the myriad other problems which beset computer users. First steps have been taken in the hardware area. The Multics system uses special hardware to free the programmer from a fixed variable-memory location number mapping [I.8]. The Basic Machine of J. K. Iliffe departs from the von Neumann architecture with its use of self-identifying data; this makes possible a memory dynamically structured to the data of the user's problem instead of a linear array [I.9].

Some steps have been taken in the software area as well. Most notable has been the development of more or less machine independent high level languages for programming. Bare machines with only loaders and stand-alone compilers and applications programs have been replaced by elaborate operating systems.

The next section of this chapter will cover some of these hardware and software developments briefly. The second chapter will present a model which may be used at
multiple levels of computer process description and analysis. The model will incorporate concepts common to all levels of computer hardware and software, from the microcode level to the operating system. The third chapter will show the results of application of the model to extant systems. It will be used to model some hardware and software systems of interest, in particular those mentioned in Chapter I. It will also be shown that several existing models for computer description and analysis may be described as special cases of the new model. Conclusions will follow in chapter IV.

EXISTING MODELS AND SYSTEMS

The modelling of computer systems has been approached from numerous directions. Among these have been (1) automata and formal language theory, (2) programming linguistics (the functional application approach taken by Landin and others), (3) switching theory, and (4) general hardware/software descriptive techniques. Some examples of models and systems which use these methods will now be given.

*Algol 60 and the Backus-Naur Form*

It cannot be denied that early algorithmic languages, most especially FORTRAN were major breakthroughs in the
development of computers as useful tools for the average user. Being the first attempts to solve the programming problem, these languages had drawbacks. FORTRAN, for example, was heavily influenced by its initial implementation on the IBM 704 (by Backus, et. al [I.10]). The language was defined as being a set of statement types, with parametric values and variables supplied by the user. The structure of the 704 was such that parameters were defined strictly as positive integers, (as in DO statement increments), integral values of the form I*J+K (as in subscripts), or simple variables (as in input/output lists). Array dimensions were limited to three since the 704 had only 3 index registers to hold subscripts. The rules and restrictions for each type of statement made the language a chore to learn. Finally, the restrictions became embedded in the language definition even though neither necessary nor convenient in other implementations. Many who have used FORTRAN compilers have found unintended restrictions and extensions of the language peculiar to those compilers. These might mysteriously forbid the use of certain variable names, or they might allow modification of the DO index within the range of the DO statement [I.11]. These may often be traced to the attempt by the compiler designer to meet the ad hoc specifications of FORTRAN.

The Algol 60 project was an attempt to overcome some of these difficulties by providing a general, machine
independent, clearly defined algorithmic language. The designers turned to automata and formal language theory, which perhaps explains the consistency and modularity of the resulting language. The Algol 60 language is context free (modulo a very few unintended ambiguities) in the sense of Hopcroft and Ullman [I.12]. The context free nature of the language requires that productions be recursive; it also encourages the usage of as few non-terminals as possible. The best example of this phenomenon is the non-terminal \textit{arithmetic expression (ae)}. The \textit{ae} appears as a statement parameter in nearly all cases, replacing the aforementioned three types, and others, found in FORTRAN. The effect is to make the language easier to learn, once one gets used to the recursive nature of the definition; it also has been to make the language fairly easy to implement. Many compilers for the language consist of recognizers for the non-terminals and terminals of the grammar which may called recursively in the order in which they appear in productions. The recognizers are usually written as sub-routines or macro instructions, and are therefore rather easier to debug than the recognizer for the context sensitive FORTRAN language.

The Algol 60 project also was responsible for the definition of the now well-known Backus-Naur Form (BNF). The BNF description of the productions of the Algol 60 grammar made the already concise language seem even more
so. The sequence of productions

DIGIT → 0
DIGIT → 1
...
DIGIT → 9

may be expressed in BNF as

<digit> ::= 0|1|2|...|9

More interestingly, the productions

UNSIGNED INTEGER → DIGIT
UNSIGNED INTEGER → UNSIGNED INTEGER DIGIT

may be replaced by the BNF construct

<unsigned integer> ::= <digit>|

<unsigned integer><digit> .

By this example, BNF may be seen to be a powerful way to represent the recursive nature of the Algol grammar. Its syntax is deceptively simple: non-terminals are placed in the < > pair. The productions are indicated by the use of the ::= . Terminals are represented by themselves and are not placed in the < > pair. The use of alternatives for a production is indicated by the use of the | . The concatenation of non-terminals and terminals indicates that these follow each other in the strings of the language.

Many implementors of the language have looked to the BNF definition of it for guidance. The syntactic recognizer approach previously mentioned seems to have had its roots in BNF productions. Recursion in BNF has been studied
prior to implementation to decide which production to apply first. It is possible to attempt to recognize <digit> first and attempt to extend it by looking for another <digit> in order to produce <unsigned integer>. This is called bottom-up analysis. Alternatively, one may attempt to compress two <digit>s into <unsigned integer> <digit>, etc. This is called top-down analysis.

While the syntax of Algol 60 is described nicely in BNF, the semantics as outlined in the Revised Algol Report [I.13] are described in English and have been open to charges of incompleteness and ambiguity, and probably were responsible for the aforementioned ambiguities of Algol 60. It has remained for the semantics of languages to be described with the ease and modularity of syntax description. This lack of notation is one of the reasons why the study of computer language semantics is not so advanced as the study of language syntax.

The ISP Notation of Bell and Newell

The Instruction Set Processor Notation (ISP) of Bell and Newell [I.14] is in essence an abstraction of register transfer languages (RTL) used for description of functional data transfers within computers. The use of formal RTL notation is in itself a fairly recent development, e.g. [I.15]. The concept, however, dates back to the earliest
days of logic design for computing machinery. Logic designers invented notation to describe logic flow in processors for their own use in the design process. This notation was often incompatible with others'; formal RTL languages were proposed to solve this communication problem. The basic concept was and is that contents of registers are transferred to other registers, and may be combined in well-specified ways with the contents of other registers in the process. Notationally, this was usually expressed as

\[ \text{ACCUM} \leftarrow \text{MEMORYDATA} \]

possibly with an operation included, as in

\[ \text{ACCUM} \leftarrow \text{ACCUM} + \text{MEMORYDATA} \]

The ISP notation seems to have been developed for two purposes: (1) to serve as a more powerful kind of RTL for the use of designers, and (2) to describe the computation process to the reader with a moderately good background in computer operations. It is in response to the second purpose that some of the differences between ISP and other kinds of RTL arise. Two of these differences of most interest are: (1) ISP is a means of describing the entire instruction set of a processor, and an interpreter to perform sequencing; a typical RTL may or may not perform the second function. (2) ISP is not oriented to optimal design or to analysis. For this reason, the resolution of the language is not so fine as would be necessary to accomplish these tasks. In their example of the PDP-8, Bell and
Newell mention that the actual instruction decoding process is not modelled in detail. What is important is the net effect and its clarity to the reader [I.16]. However, the authors point out that the structure of ISP is such that it could have been modified to be more rigorous had they so chosen.

ISP is evidently a viable language for its purpose, as Bell and Newell go on to use it to describe widely differing processors in their book. Its usefulness is definitely limited to hardware description, however.

*The Contour Model*

It was the intent of one of the developers of the contour model of block structured process description (CM) to provide a model "to be of service both in pedagogy and in machine and language design" [I.17]. The language class with which the model works the best is that of the block structured languages such as Algol 60, PL/I, and especially Oregano [I.18], among others. Its pedagogical value lies in the fact that it lends itself readily to graphical presentation. A computation is presented as a sequence of *snapshots*, of which there is usually one or more per program step. These snapshots present graphically the complex relationships between variables in different blocks of the program. The model also provides data description and
instruction definitions which are intended to be a useful
set of primitives for language definition and processor
specification.

In the literature, the usage of the contour model has
been confined to explication of language semantics by
example. Berry, for example, uses it to explain the reten-
tive nature of his Oregano language. In such invocations,
the heart of the CM is a processor, denoted by \( \Pi \). The
processor consists of two parts, the instruction pointer
\( \Pi.ip \), and the environment pointer \( \Pi.ep \).

The instruction pointer points to the next instruction
to be executed, which is located in an invariant represen-
tation of the program called the algorithm. The algorithm
description of the program is nested into contours, which
are nested regions corresponding to the block structuring
of the program. There is a contour for each block and for
each procedure. The instructions in the algorithm are in
an unspecified form, and are usually represented symbol-
ically in snapshots, a sample of which is shown as Figure
I-2.

The environment pointer is a place holder in a non-
invariant record of execution. The elements of the record
of execution are the variables of the process, plus static
and dynamic links. The static links are used to reflect
the lexical nesting of blocks and procedures. Dynamic
links are needed to reflect the changing environments
First CALL of \( P \) performed. Illustrating setting of \( B \) to 67 in block \( B2 \) of second CALL of \( P \).

Heavy line indicates current activity.

Figure I-2.
A Contour Model Snapshot.
caused by procedure invocation (See [I.19] for an excellent discussion of this problem in terms of a stack model.) It is shown that a transfer through a label into or out of a procedure may change the environment of the processor, i.e. change the set of variables known to it. It is then shown that the set of static and dynamic links allow for the correct environment to be established when this event occurs.

A vital adjunct to the linking mechanism is the use of the label as a data type. A label in the CM is not simply an instruction pointer. Rather, it is a pair (ep, ip) which establishes a new environment via ep and a new locus of control via ip. The label is then used to implement both procedure returns and non-standard transfers out of procedures. Both cases are handled in the same manner, a useful simplification of the procedure exit problem. An operation, MKLAB, is provided to construct labels from the current environment and the desired instruction number, and to check the resulting constructed label for validity. The created label is then treated as data, except that the processor automatically changes the environment when it references the label.

The use of the pointer data type is also allowed in the contour model, and it is shown that this data type presents the designer of a block structured language based on a stack model with a problem: A pointer declared in an
outer block may be assigned to point to a variable declared in an inner block. When the inner block is exited, the pointer remains set to point at the variable in the vanished inner block. This phenomenon, known as retention, is shown to cause failures when stack-oriented block exit routines pop the variable stack. The processor of the contour model circumvents the problem graphically by the judicious use of arrows, and definitionally by providing reference counts to be updated when a block is exited. The contour for a block will not be deleted until such time as the reference counts for all variables in the block are zero. The value the contour model has for describing such languages as PL/I and Algol 68 which both support pointer data types is evident. To the extent that both of these languages were intended for stack implementation, restrictions have been made in the specifications of the languages to disallow retention or to require that retained variables be allocated in special data areas. The contour model obviates the need for these restrictions. It is in this aspect that it may prove of value in language and processor design.

The formal specification of the CM consists of two parts: the cellular and static global structure, and the instruction repertoire for a processor to execute cells of programs. First, the cell is defined as being a string of boxes (bits), subdivided into two parts—the cell organization part and the residence part. The residence part
contains data or instructions; the organization part contains a cell description and a reference count. The cell description indicates what sort of cell it may be, e.g. integer, pointer, label, or instruction. It may also contain access restriction and presence information (in an actual hardware version).

A contour cell is a special cell type which contains, in addition to the previous fields, a contour valid bit, a static link, an antecedent link, and a contour height value. The static link points to the immediately enclosing contour. The antecedent link points to the beginning of the corresponding block in the algorithm part of the model. The contour height indicates the total nesting depth of the current contour, and is evidently needed to maintain the validity of a display used for inter-contour variable searches.

A processor is also a special kind of data cell, and it actually contains more than an ep and an ip. It also contains cells indicating whether it is currently active, a pointer to itself used for inter-processor communication, a site of activity register (a label describing its current location and environment), an instruction register, an instruction height register, and other auxiliary registers. It also contains a set of display registers which are used to reference variables not contained in the current contour. These are numbered from 1 to some maximum, and the ith
Figure I-3.
Before and After an ENTER Instruction.
(From [I.17], page 79).
register addresses the environment of the ith level of nesting. The registers must be updated when blocks are entered or exited, and often when procedures are invoked. A very simple example of the use of the processor registers for block entry and exit is given in Figure I-3.

The instruction repertoire consists of the instructions which allocate and delete cells, including contours and processors, begin and terminate processors, perform block entry and exit, and perform conditional transfer of control. To this set is added the operations peculiar to the language being studied.

The contour model has gained deserved recognition as a descriptive tool for the semantics of block structured processes. Though rather hazily defined initially, it should become more formal as experience is gained in its use.

The Vienna Definition Language

The formal definition of PL/I by Lucas, Walk, et. al. at the IBM Vienna Laboratory (IBMVL) was based on the definition of an interpreter for programs in the language as the defining instrument for that language [I.20]. J. A. N. Lee has extended the set of primitives developed by the IBMVL group to a system for defining algorithms and computer processors, as well as the semantics of languages, with
enough formality so that the basis for proofs about the
definitions may be laid [I.21].

The system used by both of these and others is called
the Vienna Definition Language, as it was initially devel-
oped by the IBMVL group. The system deals with primitives
called objects. A composite object is a tree-structured
entity whose branches are named by selectors, and whose
subtrees and leaves are objects. An elementary object has
no descendants and must therefore be a leaf of the tree of
a composite object. Objects may be associated with a name;
the resultant entity is a named object and is written as
<s:X>
where s is a selector and X is an object. Objects may be
represented uniquely by a set of pairs

{<ξ:eo>}

where ξ ∈ S, the universal set of selector names, and eo ∈
EO, the set of elementary objects. Objects may also be
represented in tree form as in Figure I-4. Manipulations
of objects are performed by the use of the μ (mutation)
operator. The general form for this operation is

μ(X;<ξ:y>)

which means that the result of the operation is the object
X with the ξ-component (if it had one) replaced by the
object y. This is known as generalized assignment. If
y=∅, the null object, the operation is that of deletion of
the ξ-component. If X=∅, the operation is the creation of
A = \{<s_1:e_1>, <s_2:\{<s_3:e_3>, <s_4:e_4>, <s_5:e_5>\}>\}

Figure I-4.
Graphical Structure of a Composite Object
X with value y. Predicates over objects may be written in various formats. These allow for LISP-like (McCarthy) conditional expressions used for sequencing and control in formal machines which manipulate objects.

The IBMVL and the Lee approach to language definition are similar. Lee chooses to give the syntax of BASIC initially in a BNF-like notation, whereas the IBMVL group assumes familiarity with the PL/I language. Each defines an abstract syntax for the language, consisting of a number of object definitions couched in terms of predicates over other, previously defined, objects. For example, a BASIC PRINT statement is defined by the expression

\[
\text{is-print-st} = (\langle s\text{-line-no} : is\text{-line-no} \rangle, \\
\langle s\text{-keyword} : PRINT \rangle, \\
\langle s\text{-output} : is\text{-variable-list} \rangle)
\]

It is the job of a postulated syntax analyzer and translator to convert particular programs in the language, e.g.

100 PRINT X,Y,Z

into an object satisfying the abstract syntax predicate which defines it:

\[
\{ \langle s\text{-line-no} : 100 \rangle, \\
\langle s\text{-key-word} : PRINT \rangle, \\
\langle s\text{-output} : \langle X, Y, Z \rangle \rangle \}
\]

which is diagrammed as Figure I-5. An interpreter is then defined which transforms the abstract syntactic represen-
Figure I-5.
Skeleton and Completed Abstract Syntax Representation of PRINT statement in BASIC
tation of the particular program and a representation of the input data for the run into an interpretation of the program. The interpreter is defined as a sequential machine, with state \( \xi \). Lee and the IBMVL group differ as to nature of components in \( \xi \). Lee generalizes \( \xi \) to consist of a data component, a tree structured control stack of instructions, a control work area, and a copy of the algorithm to be executed. This definition allows for the machine to model algorithms and processors, as well as the semantics of languages. State transitions occur when an instruction at a terminal node of the control tree is executed. These instructions may be of two types: (1) self-replacing, and (2) value returning. A self-replacing instruction causes the node of the control tree to be replaced by another (usually tree-structured) instruction, possibly conditionally selected by the use of the McCarthy notation mentioned earlier. A value-returning instruction returns a conditionally computed value to the parameter list of the instruction which calls it. It may also modify certain parts of the state. Its node in the control tree is deleted. These types of instructions are diagrammed in Figures I-6, and I-7.

Lee uses the VDL to describe algorithms such as addition, data structure manipulations, and sorting. He also uses it to describe the PDP-8 minicomputer. The notation and conventions used in these are much like those used in
Figure I-6.

Self-replacing Instruction and Its Effect
Figure I-7.

Value-returning Instruction and Its Effect
the definition of language semantics.

The difficulty with VDL is its complexity. The notation becomes cumbersome, even for the description of a simple process. A great number of supportive definitions, functions, and transformations are needed. This is the price of formality. For example, the ISP description of the PDP-8 is two pages long; that in Lee is fourteen pages long [I.22, I.23]. The detail presented in the Lee version is so meticulous that its explanatory value is doubtful, even though it is the more formal. It is a first iteration on the problem of computer semantics, and it is to be expected that it might not be optimal. The model to be presented in the next chapter is a second iteration, and would possibly be better. Later efforts will improve the situation further.

The Multics System

The intent of the designers of the Multics (Multiplexed Information and Computing Service) System was to provide a "computing system which is capable of meeting almost all of the present and near future requirements of a large computer utility." [I.24]. In terms of computer hardware and software, they meant a system with a large processor and file storage system, which is both highly reliable and extensible. These goals would be met by (1) virtual storage with reference by logical segment name and displacement
with the segment, removing core address restrictions from programs, (2) run time binding (linking of programs to data), (3) pure (read-only) procedures which would be shared among many users, (4) extensive automatic backup and management of the large multi-device file system, and (5) pooled usage of multiple processors, memory boxes, input/output controllers, and other devices subject to periodic failure [I.25].

The system was designed around the General Electric (now Honeywell) 645 computer, a modified version of the GE 635. The extensions to the 635 processor were made primarily in the area of addressing hardware; it was these extensions which allowed many of the advanced features of Multics. In view of the fact that these addressing features are so vital to Multics, the discussion of that system will center around them, rather briefly here, in some detail in Chapter III. (The material to follow comes from several sources, most notably Organick [I.26]).

The address space of a Multics processor is not the same as the set of physical memory locations; rather it is broken down into symbolically, dynamically extensible segments. A segment is initially referenced by name. If the segment is not in core, it is retrieved from the file system, or created, as appropriate. It is then bound to a number for the duration of its usage by a process operating within Multics. This segment number, together with an
index-within-segment offset, defines a location in the virtual address space of the computer. This location in virtual memory is mapped onto a physical core address through a segment descriptor, which describes the nature of the segment and gives information about it: its bounds, type, location, access rights, and whether it is paged. The set of segment descriptors for a process, which are collected into a descriptor segment, constitute the address space of the process in its entirety. A process may access this space and none other, due to the nature of the address formation system in the 645. Moreover, it is not allowed to address out of the limits defined by the bounds information for each segment.

Addressing is facilitated further by two more extensions incorporated into the 645: address base registers, and indirect word pairs. The address base registers are used to hold segment number-offset pairs for commonly used data for the process. These registers, used in pairs, are used normally to point to the argument list and linkage segment for a procedure, and to a processor stack. The fourth pair is used for general work. Generalized indirect address word pairs, known as indirect to segment (its) pairs, and indirect through base (itb) pairs, allow symbolic indirect addressing in addition to the symbolic direct addressing provided by the hardware mentioned above. Its and itb pairs may be used to address to any number of
levels, through a number of different segments. At each level, the reference is by segment number-offset pair, insuring full generality of addressing.

The modularity of the Multics system seems to have been induced for two reasons: ease of development, and system integrity. The system is easy to debug in that modules may be debugged separately, using contrived test data and running at non-privileged user status. When a module is shown to work, it may then replace zero, one, or more modules already in the system with little difficulty. System integrity is aided by the use of the concept of the process. A process might typically be a file editing session followed by a compilation, and then a run under an on-line debugging aid. One of the goals of Multics was to use pure procedures (i.e. invariant write-protected code) to a large extent. By breaking a process into a large number of calls on such pure routines as compilers, file managers, and debugging routines, this objective is achieved. Further, each of these separate modules may execute in a different protection ring which defines its priviledges. A file manager might need to execute in an inner, highly privileged ring; the compiler usually needs only user priviledges. Crossing from one ring to another is monitored by both hardware and software.

The Multics system has been sucessful enough to be offered for sale commercially. The features that have been
discussed here, it is felt, have contributed to this success. For this reason, they should be studied carefully.
CHAPTER II
PRESENTATION OF THE MODEL

The foregoing models have been of a very diverse nature in both concept and description. Most are also complex. The fact of the matter is that the data structures, operations, and all of the other parts of a computation are in themselves complex; descriptions of them must also be so. Models which are the most lucid in their descriptions do not attempt to deal with all of the computational process. Some restrict themselves to the domain of a single process within the computer, treating the other parts on an input-output basis. The BNF model is such a model for programming languages. On the other hand, some are capable of dealing with all aspects of a computational process and are conversely difficult, especially for the uninitiated, to understand. The VDL modelling system is the classic example.

The objective of this work is to present a modelling system with the power of the most formal model and the clarity of the most lucid and informal. After some study of the existing models, it was found that a computer system, Multics, may have as much or more to contribute to the search for such a model as existing models themselves. For this reason, it has been included in Chapter I. The Multics System offers at least two clues to the logical
design of computer models as well as computer systems: (1). It is hierarchically structured on the basis of a process. (2). Entities (segments) in the system are treated much alike in terms of primitive operations by both the hardware processor and the system itself.

It seems that the understanding of something as complex as a computer system is aided by considering the small process entities of which the system might be composed. Process modelling is more easily accomplished than systems modelling, and the resulting models are more tractable in use. In particular, a process model can be made formal enough to obtain proofs of results of interest to students of computer systems. One may construct a process model such that an induction on a set of processes models some desired whole system. That is, the system can be modelled as a set of processes, each of which may in turn be a set of processes. One need only consider the smallest of these and their interactions to grasp the whole system.

The treatment of computational entities on a structured basis runs through not only Multics, but many of the foregoing models as well. There is, however, a dichotomy in the nature of entities which not all of these models recognize. A study of the models and of the manner in which Multics handles segments indicates that a good process definition mechanism must handle the concepts of the computational object and of the named variable as two separate,
but complementary aspects of data transformation, i.e. computation. The Multics System has the facility to associate names with objects in a particularly straightforward way, a feature not found in other models and systems. A system based on the Multics facility seems to be one of the best for use in a descriptive model; it will be used in the model to be presented in this chapter.

A question which has also received some study is the structure of the state transformations which take place in computational processes. The Vienna Language interpretive system seems to be one of the better systems in this regard in its usage of what is in effect a sequential machine performing state transitions on a state tree. The formalism of a sequential machine of some sort and its underlying body of theory is needed if one is to use the model to prove results about specific types of computation. A machine to perform an equivalent function in the model to be presented in this chapter will be informally introduced in the next section. This will be followed by formal definitions of the object and the variable, which constitute the heart of the process model. The actual definition of a process modelling machine will then be given.

A PROCESS MACHINE

The Turing machine (TM) is a familiar conceptual
process. It has been found most useful as a canonical notation for expressing the primitive processes of computing, such as addition and multiplication. It is not, however, well suited to discussion of computer computation for the primary reason that a TM is presumed to have memory in the form of an infinite tape. Actual computations do not work on tapes, in general, and while they make take indeterminate amounts of memory, the amount available to them is never infinite. The usage of tapes creates a difficult access problem. In fact, a large part of almost any TM program is the movement of data on the tape in order to put it in the proper place to be transformed. In an actual computation, of course, the variables are located in a random access store when they are needed, although they may reside on serial backing store when not needed at some instant. Any variable is theoretically available instantly, though it is sometimes found useful to restrict accessing to occur via stacks, queues, or other similar systems.

The computational power of a TM is well known. A TM may be found to solve any problem of interest, although its operational primitives leave much to elegance. It is possible to design into a processor any set of instructions that might be desired, with the knowledge that each may be reduced to a macroscopic TM program. Given this fact, a process machine may be informally defined as a random access automaton. In fact, such an automaton is essentially
equivalent to the previously known stack automaton, with the additional proviso that any element of the stack may be accessed at any time. As in the case of the normal TM, the process machine is presented with an initial configuration—in memory instead of on a tape. It then performs its computation, altering its random access memory to record the problem state as it progresses. It halts with the answer located in one or more the variables in its memory.

ELEMENTS OF THE FORMAL MODEL

Objects

The two main portions of a computer are the memory and the central processor. Memory structures in previous computer designs have been very regular. In the typical case, the memory simply consists of a number of words, characters, or decimal digits, each associated with an address. The central processor as a rule is required to operate on only on of these fixed length entities. Advances in technology have made possible newer memory structures. Among these are the associative memory and finer address resolution (e.g. the byte or even the bit). However, their usefulness is limited to processors specifically equipped to handle them. It is true that a high-speed byte addres-
sable memory might be attached to an older processor for the purpose of replacing an older, slower, memory; however, the byte addressing ability would be disabled, a clear waste of resources.

Further developments in processors, and especially memories, are expected to continue. Magnetic bubble technology offers the promise of memories which may include logic to perform simple processing within the memory itself, such as garbage collection. Large scale integration may lead to processors arranged in an iterative array of identical cells (perhaps arranged as proposed by Kane [II.1]). Given the many different directions processor design might take, it is clear that avoidance of processor-memory mismatches will become ever more difficult unless a unified design approach is taken.

It is proposed that a processor and memory independent data structure system is needed in order to provide an objective means for their joint compatible design. An entity called the *object*, which is the foundation upon which such a data structure will be built, will be defined. The concept of the object is not new [II.2,II.3,II.4]; its usage in this work is a distillation of the important aspects of several definitions of it. Additionally, some extensions to the original definitions have been made. The unified definition to be presented will allow the object to be used in some useful new ways.
An object is a tree structured data item, as illustrated in Figure II-1. The terminal nodes or leaves are elementary objects. These are the primitives with which the processor deals. They may be integers, logical quantities, characters, etc. An object carries with it a code that indicates its nature.\footnote{See note in References} This type code is a vital part of the object, and is one of the extensions that have been made to the original concept. With the type field present, the processor can take action appropriate to the data presented to it. For example, it could convert integer to floating before operations. It could also escape (trap, interrupt) when it encounters unfamiliar data types, such as double precision or complex. A software routine would then handle the operation. Small computers in a manufacturer's line would truly be compatible with larger ones in the same line in that the same data structures and programs would run on either computer. In a large system, hardware would do the work that software would do in the smaller system. It would do it faster, but at greater hardware expense.

The non-terminal nodes of the object tree structure are entities not normally processed as data. A non-terminal node indicates the number and nature of its descendant nodes. The storage efficiency with which this is done varies, depending on the nature of the descendants themselves. Non-terminal nodes carry with them type information
Figure II-1.
An Object Tree Structure.
not dissimilar to that of terminal nodes. The type information of two types: (1) a direct type which indicates that the node is non-terminal and should not be processed as data, and (2) an indirect type to allow the use of non-terminals as representatives for the data at the descendant nodes. An example is given of a $3 \times 2$ matrix as it would be represented in this system as Figure II-2.

Objects are formally characterized in the following way:

**Definitions:** An object is an elementary object or a composite object.

An elementary object is an element $x \in S$ where $S = \{\text{some appropriate universe of computational values}\}$. The null object $\Omega$ is always included in $S$.

A composite object $X$ contains an index set $I$. $I = \{i \mid \text{lower bound}(X) \leq i \leq \text{upper bound}(X)\}$. $\forall i \in I$ there is an elementary object $x$ which is a component of $X$ such that $x \neq \Omega$.

The above is a rather restricted definition in that it is difficult to represent some entities of interest. Among these are sparse vectors and transposed matrices. The definition given may be extended for such cases; however, the extension must be on a case by case basis and is too
A MEMORY REPRESENTATION

Key:

A - Address field
DT - Direct Type field
IT - Indirect Type field
I0 - Initial Index field
L - Length field
VAL - Value field for data
D - Type (Direct or Indirect) is Descriptor
N - Type (Direct or Indirect) is Numeric Data

Figure II-2.
A 3×2 Matrix - Object Tree Structure and a Memory Representation.
complex to pursue in this work. On the other hand, it is possible to construct such entities as trees, matrices, matrices of matrices, and other high order structures using the object as defined above. The expansion of the set of elements of computation to include pointers will allow the formation of very complex data structures containing self references, external references, and program segments. This will be done shortly. First, further comments on the implications of the composite object are given.

The actual relationship of interest is that between self-identifying data and the hardware and software that manipulate it. A case in point is the language APL [II.5]. The definition of APL requires that a scalar quantity, a one element vector, and a one by one matrix be separate entities, and that an error is to be signaled if one attempts to use one type where another is required. It is clear that much information about variables must be retained by an APL interpreter in order to detect such conditions. When the information accompanies the data, two advantages are evident. The first is localization. The data and its description are not logically separated. Accessing information about the data structure is equivalent to accessing the data itself. The second advantage is hardware recognizability. The above case of scalars versus one element vectors would be recognized by the the hardware of processor when it attempts to perform an operation. Two vectors
which are of equal length and type could be added by presenting their root nodes to the hardware to do with as it would. Consider the example previously given regarding a manufacturer's line of computers. An inexpensive model not properly equipped would sense that the vectors to be added were non-scalar and would escape to software to perform the vector operation. On the other hand, a larger machine with a vector processing unit would perform the operation directly. Tests for equal length could be carried out before processing since length information is contained in the index sets of composite variables. A multiple execution unit machine, such as the Texas Instruments Advanced Scientific Computer (ASC), could pass the data on to a functional unit which would operate independently [II.6]. The interpreter would proceed to the next operation. Data accessing conflicts might thereby be generated. These could be easily spotted and arbitrated, however, since it is always possible to determine if two root nodes describe, or are a part of, the same objects undergoing processing. A master controller with address sensing hardware could easily perform such arbitration.

The definition of the object will now be extended to another non-terminal node class. The intermediate object is defined thus:

Definition: an intermediate object is a composite object X,
such that

(1) The index set is null.
(2) There is exactly one component object, which may be the null object, Ω.
(3) The component of X, if it exists, is a component of some other composite object, Y.

**Comment:** Intermediate objects are not normally manipulated arithmetically; they can be neither modified nor created without special system action.

**Example:** It has been suggested that for a program to execute correctly, even though the code is correct by itself, it should be entered at specific points, called *gates* in the Multics system [II.7]. The R-2 computer will use a similar device, called the *control word* [II.8]. Consider the structure given in Figure II-3. The addresses α and β are the only points through which the vector of instructions may be entered. Call validity instructions may be placed there with certainty that they will be executed for every call on the routine. The word marked as a descriptor is a composite object whose purpose is to describe the vector of instructions as part of some storage structure system. It is forbidden to use the descriptor as a jump pointer. Instead, the control words are used. The hardware, or software interpreter would expect to see a control word for any transfer out of the current block of instructions which
Key:
A - Address field
DT - Direct Tag field
IT - Indirect Tag field
I₀ - Initial Index field
L - Length field
VAL - Value field for data
D - Type is descriptor
C - Type is control word
I - Type is instruction

Figure II-3.
Protected Entry to Code via Control Words
it would happen to be executing.

Example: The indirect address word is a pointer which is invisible to a program. This is illustrated by the data structure in Figure II-4, which might have been generated by code for a typical block structured language. The enclosing structure is a parameter stack. The procedures P1, P2, and P3 are dynamically nested. An instruction in P3 calls for parameter number zero, which is a name parameter (in the Algol 60 sense). It references the parameter pointer plus zero. The indirect address is noticed by the memory control, so that the datum pointed at by the indirect address word is fetched or used as a store destination. Now suppose parameter number one is referenced. In this event, the indirection yields a parameter of the previous procedure, P2, and another indirection is used to retrieve the datum present in the P1 stack. The salient point is that the instruction in the procedure P3 did not need to know how many levels of indirection were needed to fetch its operand, nor did it have to manually trace out the addressing path. There are certain pathological cases, such as passing procedures as parameters which would require special action (See Chapter III). Even in these cases, the advantages of self-identified objects should be obvious.
Figure II-4
Intermediate Objects as Parameter Pointers
Variables

Program correctness is becoming an important issue in computer science. When programmers switched from assembly language programming to the higher level languages such as FORTRAN and COBOL, it was felt that programs would progress from the conception stage to the production stage faster and more easily than before. Many programmers even today would say that this is not the case. The reason seems to be that while the number of errors per line of code has decreased, the number of lines of code have increased. This in turn has been due to the increased power of both hardware and software systems, which have encouraged programmers to attempt heretofore unthinkable complex projects [II.9]. A typical large project involves more coding and debugging than a single programmer is able to undertake. It would be useful if a programmer could write a module of a system and prove (or at least demonstrate) its correctness. Then presumably the collection of any number of modules would be more likely to yield a correct system, provided the program interfaces were acceptable as mentioned in the last section.

It is not intended to pursue the question of program correctness further in this work. The above discussion merely served to motivate the following discussion and definition of the program variable.
The problem of non-functioning programs dates from the earliest programmable computers. In their early work with the IAS Computer, von Neumann and Goldstine were concerned with demonstrations of program correctness. Their approach was to make assertions about the set of expected values of a set of variables—a state vector—as the computation proceeded. Errors were indicated if one of the state vectors deviated from its expected value for a known input [II.10]. Disregarding the very specialized nature of the function table computations that the IAS Computer performed, and whether or not one accepts the method of proof as valid or practical, this method has two merits which should be considered. The first is the logical nature of the method. The domain of discourse is limited to entities which are directly related to the programmer's conception of the problem. No program diagrams, flowcharts, or other abstractions are required other than those of the programmer's own choice. The advantage is the fact that even neophyte programmers are able to use the method. The second merit is the hardware independence of the variable. Unfortunately, during the early and middle years of programming, which are here taken to be the years until the development of data-rich languages such as PL/I, the idea of the variable has tended to become synonymous with the machine word. Ideally, however, the two should be distinct. It is certainly clear that a word size of \( n \) bits and a variable
size of $n+2$ bits causes extra work on the part of a computer system for some language. On the other hand, some memories now deliver as many as 256 bits or more at a time, allowing variables to be any size that divides nicely into the available bit paths.

The foregoing is the heuristic basis for the definition to follow. In theory, the variable, like the object, is not new. The definition to be presented, however, is different than that which might be expected. The variable is defined as follows:

**Definition:** A *variable* is an ordered pair $(n,o)$.

where

- $n$ is a representation of a name.
- $o$ is an object or a set of variables.

If $o$ is a set of variables, then $o$ has an index set $J$, $J=0,1,...,n$. There is a 1-1 correspondence between $j \in J$ and $v \in o$. The ordering in $o$ is otherwise undefined.

**Comment:** The representation of the name of a variable may have any form consistent with the system being modelled. It may even be implicit if the system is so structured that the effect achieved is the same as if some representation of the name were retained.

The variable will be the fundamental unit of computation in a hierarchical process modelling system. It is
the link between the object at the hardware/software level on the one hand and the programmer's conception of computational processes on the other. Processes may thus be modelled by formulation of abstract computations as sequences of operations on variables. Abstract instructions will reference variables. When a variable is referenced which is a name for a pure object, the variable name will be used as the argument of a function

$$\text{Ob} : \{ \text{variable names} \} \rightarrow \{ \text{objects} \} \ ; \text{Ob}(n) = o$$

The function is supplied with a name which must be the first element of some ordered pair in the set of variables in a process. The corresponding second element of the pair is returned, if the pair exists. Otherwise Ob = Ω. This function will be different for a given instance of system being described, and considerable freedom will be allowed. This will be seen in the following examples. When the name names a set of variables, a different function is used.

$$\text{Sub} : \{ \text{variable names} \times \text{variable names} \} \rightarrow \{ \text{variables} \} \ ; \text{Sub}(X,Y) = X.Y$$

**Example:** This example is a general case. In implementation, it would require either time-consuming extensive software support or hardware support in the form of a very large associative store. The variable names for a process are retained in (some representation of) their alphanumerical form. When a reference to a variable is made in some context, e.g.
ADD Q3

The name Q3 would be looked up in the name table. The result of the search would be (a pointer to) the object part of Q3, and the data fetch (in this case) would proceed. This scheme is general in the sense that the name "Q3" is not bound to a given object in any static way. Q3 could be a real number at one instant, and an integer matrix in the next. As far as is known, this scheme has never been implemented completely\(^2\), though a primitive form of it existed on the Rice Research Computer (R-1) [II.11], which used a run-time symbol table to perform name-address binding when programs were loaded. The use of indirection through the symbol table allowed dynamicity of reference. It was the responsibility of the user or the compiler to keep track of the changes, however. The system was very successful, and was considered by its users to be very advanced for its time (circa 1964-71). An interesting extension which has apparently never been used except in interpretive systems such as LISP and SNOBOL is the use of another elementary object type—the "variable name". A reference to Q3 in this case would result in two name searches. This would occur since the result of the first search would be determined to be other than an arithmetic quantity. Unlike Multics (described below), binding of variables is never finalized.

Example: The next step below such a general system is the

\(^2\)See References section
Multics system, as described by Organick [II.12] and briefly reviewed in Chapter I. In this system, a variable reference is retained in symbolic form until the name is first used. At that point, the name is bound to an object called a segment for so long as the program is in execution. This is done much as in the R-1 system, by the use of indirection. A pair of words in a special data area addressed by a machine register are used to first hold the symbolic name, and later the indirect address word pair when binding is finalized.

**Example:** The least sophisticated variable reference systems are found in connection with high level language programming systems. The compiler provides the software support necessary to map all, or nearly all, references from names to objects. Even block structured languages normally make use of variable name maps which are finalized at compilation time. Those references not completed by the compiler are normally completed by a process known as linkage editing. Some languages where the variable name mapping is more dynamic are not suitable to such treatment, and the language implementor is faced with a formidable task in designing his own flexible binding system. This usually entails a great deal of work, and may have a chilling effect on the acceptance of such languages. A case in point is Algol 68, which five years after its design had not been fully implemented anywhere [II.13].
Notational Conventions

This section gives some notation which will be used in later sections of this work. It will serve as a basis for a data description system to be used in connection with a formal definition of the language BASIC in Appendix B.

(1) A variable which is composite and indexed, i.e. of the form

\[(\text{var}, \text{object}) \text{ where object is composite} \]

is indexed by the integers in the index set. For example, the fourth component of variable \( M \) is written \( M.4 \).

A variable of the form

\[(\text{var}, \{V_{i_1}, V_{i_2}, \ldots V_{i_n}\})\]

is not indexed, except that an implementation may associate a mapping to the indices \( i_1, i_2, \ldots i_n \) for convenience. This has nothing to do with the conceptual ordering of the component variables. The components are selected by name e.g. in the variable

\[(X, \{Y, Z\}) = (X, \{Z, Y\}) = (X, \{(Y, 3), (Z, 4)\})\]

the Z-component is addressed as \( X.Z \).

(2) In the case where the numerical value of a component of a variable which is in turn a variable is needed, the name of the component is enclosed in parentheses.

For example, \( X.Z \) is a variable, but \( (X.Z) \) is an object whose value is 4. This object may, of course, be used as an index for an indexed composite variable. In the example
above, M.4 means the same as M.(X,Z).

(3) The notation \text{X.[Y].Z} indicates that the variable or expression in the brackets is a variable whose value is in turn a name to be found and substituted. If the value of Y is "Q", then the reference above is in fact to X.Q.Z.

(4) The name of a variable may be used interchangeably with the parenthesized notation given in the definition of a variable. In particular, the following notation will be used a great deal:

\text{VAR=(VAR,17)}

This means \text{(VAR,17)} is meant when VAR is mentioned.

(5) A variable may be specified by several names in the case where it is a component of a component of a component, etc., of a variable. Notationally, dots are used to indicate component selection, \text{e.g. X.Y.Z.A.B.C}.

(6) The highest level qualifiers for a variable name need not be written explicitly if they may be understood to be present. In the example above, discussion of A.B.C could be carried on if it were known that X.Y.Z. was implicitly written before A. Similarly, a variable need not have the lowest level qualifier set added if it is understood or of no interest. In the above example, X is a variable suitable for discussion, as is X.Y, as is X.Y.Z, etc., depending on the context.

(7) The names of variables are written in upper case. When a notation such as \text{BLOCK.namei.Q} is encountered, it
should be taken to mean that something satisfying conditions attached to namei is to be written in place of namei in actual use.

(8) Variables may be diagrammed as the objects which are their components, with the name beside the object. This extends to components which are variables. Examples appear in Figure II-5.

THE PROCESS MACHINE

Using the definition of the concepts of the object and the variable, it is now possible to proceed with a definition of the process machine. The model as it will be defined will be a schema for several models for the same process operating at various levels. A set of variables may be defined with certain relationships that hold among them. By changing the sequencing rule, the model may be made to execute at differing levels of description either sequentially or simultaneously. For example, the description of a computer process may be described simultaneously at the microcode and operating system level using variables which are related by set inclusion and different sequencing and transition rules. The entirety of the description is the model; it is an integration of several related models. Each sub model is therefore called an i-model.

Before the model is actually presented, some notational
Figure II-5.

Representations of Variables - Two Types.
conventions and definitions are given.

(1). Let $N$ be the set of all possible names for a variable in some system of interest, and let $O$ be the set of objects in the same system. $N$ is finite, owing to the fact that the members of $N$ must be representable within the computer as parts of variables. Let $n$ be a subset of $N$. Define $V^*$ as the set of variables produced from $n \times O$.

Define $V$ as a subset of $V^*$ such that for every pair $(n_i, o) \in V$, $n_i \neq n_j$ if $i \neq j$. $V$ is a subset of the possible variables such that each name in $V^*$ appears exactly once and is associated with one object. $V^*$ is finite since the representation of $O$ within a finite computer is arbitrary but finite for every $o \in O$. $V^*$ will be called the operational closure of $V$.

(2). Let $F: D \rightarrow R$

be a function mapping elements $d \in D$ into elements $r \in R$. Then define $F(D)$ in the obvious way—that it takes the entire domain $D$ into a subset of $R$. That is, if $F(d) = r$, then let $F(D) = R^*_D$.

The Definition of the Processor

The definition of a process machine is now given. A process machine is defined as a 6-tuple:
Definition: \( P_A = (V_0, V, V^*, \sigma, F, H) \)

\( V_0 \in V^* \) is the initial state of the process.

\( V \) is the state variable set (state vector).

\( V^* \) is the operational closure of \( V \).

\( F \) is the input transformation.

\( \sigma \) is the state transition function.

\( H \) is the output function.

\( F \) corresponds to the instruction fetch cycle of a normal computer. In general, \( F \) is defined in the following way:

\[
F : V^* \to O ; \quad F(V) = o \in O
\]

where 0 is the set of all objects representable in the system of interest. \( F \) operates on the current state variable set of the process and yields a (possibly composite) designated variable which is the instruction to be executed.

\( \sigma \) is the state transition function, defined in general as:

\[
\sigma : V^* \times F(V^*) \to V^* ; \quad \sigma(V, F(V)) = V'
\]

The state transition function takes the current state variable set \( V \in V^* \), and the fetched instruction \( F(V) \) to produce a new state variable set \( V' \in V^* \).

The output function \( H \) produces as output the set of variables changed by the state transition. Its domain and range are:
\[ H : \mathcal{V} \times F(\mathcal{V}) \rightarrow \mathcal{V}. \]

The current state and instruction are examined. The instruction is executed, and the variables on which it operated are output, together with their new values. A computational process \( P \) may be reconstructed by examination of \( H(P) \).

**Discussion**

The initial state \( V_0 \) is a set of variables with which the process begins life. It is specified that \( V_0 \) is non-empty. In particular, it must contain the semaphores, status words, and values to be loaded into relevant registers including the instruction variable when the process begins execution. There is nothing which has gone before to indicate that the process could not reference a variable which is not in \( V_0 \). To prevent the possibility of referencing such an uninitialized variable, it is possible to define a new \( V'_0 \). This \( V'_0 \) consists of \( V_0 \) and, for every name not defined in \( V_0 \), the variable \((n, \Omega)\). Attempting to reference the null object may be defined to cause an escape action.

\( V \) is the state variable set of the process in that it contains all that is needed to completely specify the process at any instant. \( \mathcal{V}^* \) as the operational closure of \( V \) is the set of all possible states for the process. In a purely abstract sense, \( \mathcal{V}^* \) would have to be considered as
infinite. It is in fact finite, for two reasons: (1) The quantities dealt with in mathematics are integers, of which there are a countably infinite number, or they are real or complex, of which there are uncountably many. Inside a computer, the representations of these quantities are undeniably finite, and are usually strictly limited in their range. (2) If input-output is to be modelled, its indeterminate nature must be considered. Input and output are defined to be shared variables. Assignment into these by the process or by some independent agency will be the input/output operation. (Functionally, this achieves the effect one thinks of when considering input/output. The only difference is that the process sends or receives data from a slightly different memory than normal.) Notwithstanding the independent nature of input/output, however, the state set of the process is still finite, because of the realizability of the process—a process may neither receive an infinite amount of data, nor produce an infinite amount of output in finite time.

The function F corresponds in its nature to the input string fr a typical finite state machine (FSM). Given the current state V of the process, the F-function selects an object. This object is the actual input to the state transition function and is held in a variable called the instruction (INSTR,F(V)). Some subfield selection operators on INSTR are usually defined, which correspond to instruc-
tion word subfields in the case of hardware simulation. In one instant, the F-function may select an ordinary machine instruction which may be executed immediately. In the next instant, F may select a control element which would begin the execution of a new procedure. The definition of the object makes it possible to distinguish between the two cases. In still another instant, F may fetch a pointer to a string which has been input to the process as a command. Again, the nature of the definition of the object will enable the control mechanism to deal with the case. What might happen is that the string is not marked as an instruction or a control pointer, with the result that the control would escape to software for interpretation of the command.

The H-function corresponds to the usual output sequence of an FSM of the Mealy type. It was defined with the goal in mind of producing as much information as the Wozencraft and Evans Model [II.14], which requires that an entire new storage be produced for each state transition [II.15]. This seems to be more than is needed, since the H-function, by retaining the changes in the variables in the storage, produces a record from which the computation may be reconstructed exactly. The tradeoff comes when the state must be known at the k-th step of some computation. The Wozencraft and Evans model produces the information immediately, in the form of the k-th storage, whereas in an i-model the H-values must be used to simulate the action of the process
step by step from 0 to k. Alternatively, an auxiliary list of all variables in V must be maintained and duplicated at each step by H. Thus, the primary usefulness of the H-function is in the demonstration of model equivalence. One definition of equivalence for two sequential machines is that they exhibit the same output for all input sequences [II.16]. This definition may be informally rephrased to state that two computations are equivalent if their H-functions produce the same values for all values of F(V). It may be further stated that two models are equivalent if their H-values may be derived from each other. This is needed for the case when variables are related by set inclusion. One model may have more variables than another, but the values for the common variables are the same at corresponding steps in a computation, and the values of the addtional variables may be computed from the values of the common variables of the two models.
CHAPTER III
RESULTS AND EQUIVALENCES WITH OTHER MODELS

Some aspects of the application of the model to various situations will be presented in this chapter. The first section will contain general comments on the invocation of the model. This will be followed by specific examples in which the relation of the model to other models such as ISP, the Contour Model, and VDL is explored. The model will also be applied to description of specific systems such as Multics.

ON THE INVOCATION OF THE I-MODEL

The proofs in this chapter do not utilize the structure of the model as it was originally intended. The model is a vehicle for comparing systems and as such should use the H-function for proving results. The proofs in this chapter are comparisons of the i-model to other models, whose equivalent H-functions are often not easily isolated. Therefore, one should not forget about the H-function merely because it does not appear in this work. It will be important in the application of the i-model to systems other than those considered here.

The proofs themselves often follow the scheme of proof which actually use the H-function, however. That is, there
must usually be an induction on the number of instructions executed in the model. This is necessary due to the equivalence definition requiring all inputs F to be used before equivalence may be established. The only means of dealing with this problem is to leave F unspecific at each step and eliminate cases on a case by case basis for the induction.

The proofs in this chapter also require that the i-model system be defined in each case so that it parallels the model being examined. It should be pointed out that this is not necessarily the method which would be used in actual practice. In other words, an i-model for block structured languages might or might not be defined as it is in the equivalence proof for the Contour Model.

Finally, a word should be said about the term "equivalence proof". The results are not equivalence results in the sense that the i-model is proposed to have an equal amount of power as the model being investigated. The term should be stated to mean that any result which can be obtained in the other model may also be obtained in the i-model which is defined in the equivalence proof. This particular i-model might or might not use all the facilities of the i-model system itself.

THE ISP NOTATION

This section concerns the relation between the inte-
grated model and the ISP notation of Bell and Newell, which was introduced in Chapter I. Since "a processor is completely described by giving its instruction set and interpreter in terms of its operations, data types, and memories, [III.1]" the equivalence between these quantities and the parts of an i-model should be defined. In doing so, use will be made without further citation of some of the examples used by Bell and Newell [III.2].

The concept of type as it is used in the definition of the object in Chapter II is much in agreement with that of the ISP data type. In order to guarantee equivalence, there must be as many type code possibilities for objects as for their equivalent data types in the ISP description of a processor. These are encoded into the object parts of the variables to be used in an i-model description of a processor.

Memories in the ISP system are also much like the concept of memory in the integrated model. Machine registers, core memory, and auxiliary memory are treated much alike in both. Were it not for the fact that ISP entities may be concatenations or subfields of other entities, it would be sufficient to establish a one to one correspondence between the memory components of ISP and the variables of an i-model. The presence of this possibility requires that there be extra variables created for these entities. It is also required that special attention be paid to the
structure of composite and intermediate objects. The object type "A" in figure III-1a indicates that it has two components, that its starting bit number is 0 and its ending bit number is 12 (Non-binary machines are not considered. To add other types would require complicating the model and proof structure. It should be fairly obvious how the model is extended to decimal or mixed binary/decimal machines.). The object type "I" is an intermediate object and contains only pointers to data. The object type "X" is composite with one component, and three bit number fields--for the starting and ending bit numbers and the starting bit number to be used from the data to which it points. These last two object types are illustrated in Figures III-1b and 1c respectively. The following procedure using these types will establish the procedure used to map ISP definitions onto variables:

(1). The Mp, the primary memory, is a composite variable MP which is indexed and whose range is that of the address space declared for Mp. The size and type codes of its components match those of the individual cells of the Mp.

(2). For simple definitions of the nature of

\[ \text{AC}<0:11> \quad \text{the accumulator} \]

a variable of the same name, type, and size is created, as in Figure III-1a.

The cases
III-1a. Simple case

III-1b.
Concatenation

KEY:
T - type field
S - starting bit number
E - ending bit number
L - local starting bit number
VAL - value
A - address

III-1c.
Subfield selection

Figure III-1.
I-Model Equivalents for ISP Entities.
LAC< L,0:11> := LAC,
which is the concatenation of two (or more entities), and
page_address<0:6> := instruction <5:11>,
which is subfield selection are treated as in the Lemma to
follow, which is summarized briefly:

Concatenations of more than two items at a time are
reduced to sequences of concatenations of two items. For
concatenations, new composite variables are created whose
components reference concatenated variables. In the case
of subfield selection, the X-type composite variables are
used. A new variable for the new entity is created; the
description part of the variable is set to reflect the size
of the smaller item. Some examples are shown in Figure
III-1b,c.

Memory equivalence Lemma

Lemma: Let \{ X_i \} be a sequence of ISP definitions of memory
entities such that X_i is not used in the definition of X_j
if i \geq j. This sequence may be transformed into a set of
variables of an i-model.

Proof: The proof will be an induction on i, the length of
a sequence \{ X_i \}, of ISP memory definitions.

Basis: The first element of \{ X_i \} must be the definition
of a simple entity, due to the definition-before-usage
restriction of the hypothesis. Therefore, ISP entity X_1
becomes a variable by direct transformation. Its name is that of the ISP definition. Its size and type bits are derived from the ISP definition in the obvious way as illustrated in Figure III-1a.

**Induction:** Assume that i statements in the sequence of ISP definitions have been processed. The i+1st transformation must be derived. There are three cases:

**Case 1:** Definition of a simple entity not in terms of previously defined entities. The transformation mentioned in the basis step may be used.

**Case 2:** \( X_{i+1} \) is defined as the concatenation of \( X_j, X_k, j, k \leq i \). ISP allows more than two entities to be concatenated. This case may be reduced to a sequence of definitions of concatenations of two entities without loss of generality. Form a new composite variable whose components are intermediate objects which point to the object parts of variables corresponding to \( X_j \) and \( X_k \). The nature of the composite object is such that the pointers must be followed to the concatenated entities themselves to find the size of the new entity by examination. The name field of the variable is given the name from the definition of the ISP entity. This is illustrated in Figure III-1b.

**Case 3:** \( X_{i+1} \) is defined as a subfield of \( X_j, j \leq i \). A new variable is created. It is marked as an X-type composite object and points to the variable of which it is a part. Its length field is set to indicate its size and its rela-
tive location in the other object. If this object is in turn a composite object, the length interpretation is carried to further levels automatically. This is indicated in Figure III-1c.

For any sequence of definitions in ISP, the above induction indicates how they would be transformed into i-model objects.

**Equivalence Result**

The complete transformation from ISP to an equivalent i-model will now be given as a Theorem.

**Theorem:** Let S be the ISP description of a processor. There is a transformation which takes S into an equivalent i-model.

**Proof:** The proof is constructive. The transformation is developed.

1. *Data types* were shown to be representable in the type fields of objects.
2. The previous Lemma showed that declaration sequences for the *memory* elements are transformable to an i-model set of variables.
3. The *operations* of ISP are made a part of the state transformation function of the i-model.
4. The *interpreter* is transformable to the fetch function
and the state transformation function of the i-model, using the operations of (3). This is done in the following way:

   Execution begins at

   instruction_execution := ( and ends at the corresponding ). Statements within the
   the parentheses are fetched and executed in sequence. If a
   statement is unconditional, it may be executed at any time, but only once per execution of the enclosing () pair. If a
   statement is conditional, the statement may be executed at any time the condition part is true, but only once per
   enclosing () pair. In either case, if the construct "next" occurs within the () pair, all possible statements in the
   () and between the preceeding "next" or "(" and the current
   "next" are executed before any after the "next".

   The above statements have been left in informal form simply because to define a formal transformation between
   the sequencing of ISP would not be particularly instructive, and would be longer than size considerations permit in this
   work. Suffice it to say that a variable which represented data for input to the state transition function could be
   formed from the corresponding ISP sequence notation.

   To the extent that ISP is formally defined, the above
   construction of an equivalent machine would perform the
   same operations as an ISP description of any processor.
THE CONTOUR MODEL

In this section an attempt will be made to establish an equivalence between the Contour Model (CM) and an i-model.

There are two approaches which might be taken when showing an equivalence between the contour model and the process model. The first is to use the same procedure as was used to model ISP, i.e. to generate the equivalent data structures as objects and show how the instruction primitives could be made into a state transition function. This could be called the direct equivalence; it is not as strong as the second method, which would be to generate an i-model with an independent structure which behaves in a manner equivalent to some corresponding CM. An attempt will be made to pursue the second method. The problem is complicated a great deal due to the haziness of the original Johnston paper defining the contour model and which serves as the principal reference for it [III.3]. Fortunately, Johnston's work is more fully explicated in Organick and Cleary [III.4] and in a later paper by Johnston [III.5]. Reference will occasionally be made to these latter papers to clear up some points.

The plan for the demonstration of equivalence is to define a set of variables in a contour model for a system
and a corresponding set for an i-model. This set is chosen to be the variables used in the program in the higher level language to be modelled. The program will be simulated by both models, using a common algorithmic representation. Instructions will be fetched from the invariant program representation and used to transform variables or to change environments (sets of variables known to the processor simulating the program) in a dynamic record of execution. It will be shown that:

(1). The operations which change the environments: block entry, procedure call, block exit, procedure return, and transfer of control result in the same changes to the designated set of variables as determined by inspection in the CM and by the H-function in the i-model.

(2). For the operations which do not change the structure of the environment, (i.e. the operations of arithmetic and assignment to existing variables), the arguments are the same and yield the same values.

The set of variables for the equivalence proof is illustrated in Figures III-2 and 3. The proof will assume that the program to be simulated by the models is located in the variable ALGORITHM, which is a composite variable. ALGORITHM.INST is a composite variable containing the instructions to be executed in an invariant form, which for convenience sake will be taken to be symbolic. The instructions which change the structure of the environment are of
Figure III-2.
The algorithm.
Figure III-3. 

The RECORD.

Note 1. Executing in block B2 within Pl called from Bl.

Note 2. Bl.EP, etc. are compressed from 2 pointers into 1.

Note 3. RETL.EP, RETL.IP are compressed into one location.

Note 4. The dotted line is an intermediate object used as a parameter pointer.
the form OP(x), and may be one of the following six instructions:

ENTER(blockname), which causes entry from the current contour into a new contour nested one level deeper in the program. An example is the ENTER(B2) instruction appearing in Figure III-2.

CALL(procname,arglist), which causes a procedure contour to be entered and arguments to be set up and placed into the formal parameters in the procedure contour.

EXIT, which causes the processor to leave the current block and to re-establish the environment in use before the current block was entered.

RETURN, which causes a return from a procedure, and which may have the effect of returning a value in addition to restoring the environment as in EXIT, above.

GOTO(label), which causes a (conditional) transfer of control to the location specified by label. It should be noted that the GOTO may cause a different environment to be entered.
Instructions are composite objects which have two variables as components. The first component of the ith instruction, ALGORITHM.INST.i.IN, is the instruction itself. The second component, ALGORITHM.INST.i.SUCC, is an integer which contains the designator of the instruction to follow the current instruction. Following Johnston, positive integer indices into ALGORITHM.INST will be used to fetch instructions.

The instructions which do not manipulate the environments and which are defined so as not to cause new environments to be created are of the form OP(X,Y). These will be defined as needed.

The instructions for the algorithm are contained in ALGORITHM.INST; the structure information is contained in ALGORITHM.STRUCT. Information from this variable is copied into RECORD, the dynamic record of execution, where it is updated as the computation progresses. The information in STRUCT is a set of variables

\[
\text{STRUCT} = (\text{STRUCT}, \{B_1, B_2, \ldots B_n, P_1, P_2, \ldots P_m\})
\]

where for convenience blocks have been given names beginning with a "B", and procedures have been given names beginning with "P". Bi and Pi have the structure

\[
\text{Bi} = (\text{Bi}, \{\text{EP}, \text{HT}\})
\]

STRUCT.Bi.HT is an integer indicating the static lexical level of the contour Bi. For example, in Figure III-2,
procedure Pl is at level 1, so that STRUCT.Pl.HT=1. The contents of STRUCT.Bi.EP or STRUCT.Pi.EP are templates. A template is a variable, which in STRUCT has the value Ω, while in RECORD it has whatever value has been assigned to it, consistent with the type information contained in the template. There is a variable in STRUCT.Bi.EP and RECORD.Bi.EP for every variable declared in the program, plus the following additional variables which embody the dynamic structure of the program in the record of execution:

SEP, the Static Environment Pointer to the statically enclosing block or procedure. For example, B2.SEP would point to B1.EP, as illustrated in Figure III-3.

DEP, the Dynamic Environment Pointer, which indicates the dynamically preceding block or procedure. Bi.DEP would parallel Bi.SEP in a language which did not allow procedures. However, in languages which do, the dynamic pointers for block activation are not the same as the static pointers. See Figure III-3.

RVAL is included if a procedure returns a value as a function. The value computed by the function is left in this variable and retrieved by the call/return sequence.
RETIL is included in all procedures. This variable is a label (a label is defined later), and is used to return from procedures. When finalized (at procedure call time), it is a label pointing to the instruction following the call to the procedure.

The i-model processor is a variable, which is also a part of RECORD, with the following structure (see Figure III-4):

\[
\text{PROC}=(\text{PROC},\{\text{IP,CEP,NEP,HT}\}), \text{ where}
\]

\[
\text{IP}=(\text{IP, index of next instruction in ALGORITHM}), \text{ the Instruction Pointer.}
\]

\[
\text{CEP}=(\text{CEP, \{variables in current contour\}}), \text{ the Current Environment.}
\]

\[
\text{NEP}=(\text{NEP, \{variables in next contour\}}), \text{ the Next Environment.}
\]

\[
\text{HT}=(\text{HT, the Current Lexical Level}).
\]

*Data Types*

Variables in the record of execution, RECORD.Bi or RECORD.Pi may be of four types:

Integers are represented as themselves.
Pointers are represented as intermediate objects pointing to the item which they reference, e.g.,

(var,pointer to object)

Label skeletons are represented as composite variables whose components are:

(var,\{EP,IP\}), where

EP=(EP,Ω), and IP=(IP,integer)

Variable var.EP is changed from Ω to a pointer to the current environment when the label is finalized.

Formal parameters are initialized to Ω. At CALL time, they are to be finalized as the values computed from parameter evaluation for call by value, by intermediate objects (pointers) for call by reference, and by labels for THUNKS in the case of call by name. Thus, the finalized parameters have one of the forms

(parm,value), (parm,pointer), or (parm,label).

The Equivalence Machine

An I-model for simulation may now be defined as the following 6-tuple:

\[ P_{cm} = (V_0, V, V^*, \sigma, F, H), \text{ where} \]

\[ V= \{ALGORITHM, RECORD\}. \]
$V^*$ is the operational closure of $V$.

$V_0$ is defined as

$$(\text{PROC}, \{(\text{IP}, 0), (\text{CEP}, \Omega), (\text{NEP}, \Omega), (\text{HT}, 0)\})$$

$F$ is defined as

$$F(V) = (\text{INSTRUCTION}, \text{ALGORITHM\_INST} \cdot (\text{PROC\_IP}))$$

$\sigma$ is defined as

$$V' = \sigma(V, F(V)) = (\text{DECODING \_INSTRUCTION})$$

$$= \text{ENTER}(\text{Bi}) \rightarrow \text{ENTERX}(\text{Bi}),$$

$$= \text{CALL}(\text{Pi}, \text{args}) \rightarrow \text{CALLX}(\text{Pi}, \text{args}),$$

$$= \text{EXIT} \rightarrow \text{EXITX},$$

$$= \text{RETURN} \rightarrow \text{RETURNX},$$

$$= \text{GOTO}(\text{label}) \rightarrow \text{GOTOX}(\text{label}),$$

$$= \text{BINOP}(X, Y) \rightarrow \text{BINOPX}(X, Y),$$

else undefined).

where the names ending in X are sets of transformations for each type of instruction to be executed. The BINOP stands for any of the non-environment-changing instructions.

The instructions which change the environment are now defined as sequences of subtransformations of $\sigma$. Notationally, they are quite simple, and are informal where they may be so without loss of generality and where an informal description is clearer than a formal one. Sequencing is indicated by the numbers to the left of the steps.
ENTERX(x):

1. PROC.NEP + ALLOCVAR(ALGORITHM.STRUCT.x.EP). This operation creates space for the variables in RECORD that will be copied from ALGORITHM.STRUCT.x, and makes it part of PROC.NEP.
2. PROC.NEP.HT + ALGORITHM.STRUCT.x.HT
3. PROC.NEP.SEP + ESEARCH((PROC.NEP.HT)-1)
4. PROC.NEP.DEP + PROC.CEP
5. PROC.CEP + PROC.NEP
6. PROC.HT + PROC.NEP.HT
7. PROC.IP + INSTRUCTION.SUCC

CALLX(x, args):

1. PROC.NEP + ALLOCVAR(ALGORITHM.STRUCT.x.EP). This is as above.
2. For all formal parameters y called by value, PROC.NEP.y + SEARCH(the actual parameter name). Note that the environment for the search is the calling environment.
3. For all actual parameters y called by reference, PROC.NEP.y + POINTER(SEARCH(the actual parameter name)).
4. PROC.CEP + RECORD.x.EP change the environment so that the environment search of dynamic links will find the correct contour.
5. PROC.NEP.RETL ← MAKELABEL(CEP, INSTRUCTION.SUCC);

create the return label.

6. PROC.NEP-HT ← ALGORITHM.STRUCT.x.HT

7. PROC.NEP.SEP ← ESEARCH((PROC.NEP.HT)-1)

8. PROC.NEP.DEP ← PROC.CEP

9. PROC.CEP ← PROC.NEP

10. PROC.HT ← PROC.NEP.HT

11. PROC.IP ← ALGORITHM.INST.x

EXITX:

1. PROC.NEP ← PROC.CEP.DEP

2. PROC.CEP ← PROC.NEP.

3. PROC.HT ← PROC.NEP.HT

Note that the contour is not to be deallocated and
that therefore pointers to variables of inner blocks
are still valid, fulfilling the retention specifi-
cation.

4. PROC.IP ← INSTRUCTION.SUCC

RETURNX:

Return is equivalent to GOTO(PROC.CEP.RETL) for the
procedure, except that the return value, if any, is
assigned to RVAL first. Thus, it is of no real interest
to define it here.

GOTO(x):
1. PROC.NEP = x.EP
2. PROC.CEP = PROC.NEP
3. PROC.HT = PROC.NEP.HT
4. PROC.IP = x.IP

Auxiliary Functions

Several auxiliary functions have been used in the above definitions. These will now be defined.

ALLOCVAR(xep):

Informally stated, ALLOCVAR is given the address in ALGORITHM.STRUCT of an environment template. It allocates sufficient space in RECORD to make a copy of xep and does so. It constructs a descriptor for the resulting copy and returns it as a return value.

ESEARCH(level):

This is a function which searches the dynamic environment chain of the process for a contour whose lexical level is equal to level. The method of search may actually be defined in other ways, and Johnston gives some of the possibilities in [III.6]. The definition to be used here is the AH (Absolutely Highest) algorithm. A situation for an environment search is illustrated in Figure III-4. The sequence of actions of ESEARCH is:

1. If level <0 then fail. Arguments must be posi-
Figure III-4.

Divergent Static and Dynamic Environment Pointers.
tive.
2. TEMP + PROC.CEP ; TEMP is a temporary variable
3. If TEMP-HT = level then return TEMP
4. Otherwise, TEMP + TEMP.DEP; go up the dynamic environment chain
5. Go to 2.

The procedure will always terminate since the argument has been restricted to integers ≥ 0 and any chain of environments must begin at the 0th (outer block) level.

SEARCH(name):

This function searches for the variable name in the (static) environment chain. Like ESEARCH, SEARCH may be specified in some manner consistent with some implementation, e.g. a display. It will be defined here as used in a chained environment.

1. TEMP + PROC.CEP ; search the current environment first.
2. If TEMP = Ω, then return Ω.
3. If name is in the environment addressed by TEMP, then return a pointer to it.
4. Otherwise, TEMP + TEMP.SEP; search the lexically enclosing block.
5. Go to 2.
POINTER(x):

For functions which return a pointer to a value, the POINTER function must be used as a special system routine to construct an additional pointer which may be stored. That is, a second level of indirection is needed to get the address of x rather than x itself.

MAKELABEL(ep,ip):

MAKELABEL constructs a label variable from the input arguments after checking that ep ≠ Ω and that ip is within the range of allowed values for the program.

Equivalence Lemma and Proof

It will now be shown that the above definition of an i-model for simulation of a program is equivalent to a corresponding contour model which uses the same environment search algorithm.

A Lemma establishing equivalence of variable accessing will first be proved separately.

**Lemma**: SEARCH(x) returns the same value or fails when the corresponding function in the CM does.

**Proof**: The proof is by induction on the instructions executed which change environments. The instructions which do not change environments need not be considered.

**Basis**: The first such instruction must be ENTER(B1), where
Bl is the outer block. Before the instruction is executed, the SEARCH function will return $\Omega$, since no environment is in existence. After entry to Bl, the functions are still equivalent since both will first search the Bl environment first. If the variable is found, both return it. If the variable is not found, both functions fail because there is no predecessor to Bl.

**Induction:** Assume that $k$ instructions have been executed and that the search functions are equivalent for all variable accesses in the first $k$ instructions. Evaluate the environment changing instructions:

**Case 1:** ENTER(Bi): The difference the ENTER instruction makes for the environment searches is that variables declared in Bi with the same names as variables declared in enclosing blocks will be found first by both models after the new contour Bi is entered. Otherwise, the situation is unchanged.

**Case 2:** CALL(Pi, args): Parameters are bound in the kth environment, known to be correct, before control passes to Pi and a new environment is used. Parameters are referenced by formal parameter name within the procedure contour and not by the name by which they are known in the contour of the calling procedure. Therefore, references to the formal parameters will be found by both search functions, which will search the procedure contour first; references to other variables will be found by the same procedure as
before procedure entry, except that a new environment
chain might be used. The equivalence will be preserved if
the environment chain search functions find the same en-
closing contour.

Case 3: EXIT: The environment entered after the EXIT instruc-
tion is one which has been used before. The effect of the
EXIT instruction is to cause the enclosing environment to
be re-established. The only difference between this environ-
ment and the previous one is that some variables are no
longer available to the search functions. However, the
same set of variables has been removed from the domain of
both functions.

Case 4: RETURN: This is much like GOTO, below. The label
used for the GOTO is PROC.CEP.RETL, which was established
in the calling environment in the first k steps of the
program, and is correct by the inductive hypothesis.

Case 5: GOTO(label): If the label given to the GOTO function
is valid, then the ep part of the label was constructed
from a valid environment where the equivalence held. The
label must have been constructed by MAKELABEL (in the
i-model) or by MKLAB (in the contour model), both of which
check environment validity when creating the label.

Since these are the only instructions which change
the environment, and thereby the domain of the search
functions, it is clear that the functions are equivalent as
defined.
Theorem: Let ALGORITHM be a program representable in the contour model. Let $X_{cm}$ be the set of variables declared in ALGORITHM. There exists an i-model such that its variables, $X_{im}$, are equal to $X_{cm}$ for every step of the computation.

Proof: The proof is again by induction. This time, it is on the set of all instructions executed in ALGORITHM.

Basis: The first instruction to be executed must be ENTER(B1). As shown in the proof of the Lemma, the variables are the same in this case.

Induction: Assume $k$ steps have been executed and the variables have been the same in each model for these $k$ steps. The next instructions may be either of the two classes previously defined—those which change the environment and those which change only the variables of the environment.

It was shown in the previous Lemma that the access functions are equivalent. If the instructions which change only the variables of the environment are the same, then the values of the variables after the new instructions are executed must still be same, by the equivalence of the access functions.

Much the same argument as that used in the proof of the Lemma holds for instructions which result in a new environment. Attention must be paid, however, to the nature of the environments set up in the cases of procedure
CALL and of GOTO, as these cause the static structure and the dynamic structure of the program to diverge. The GOTO definition which has been used requires that the contour containing the destination has been visited, i.e. that MAKELABEL finds a current environment to use for the label ep. If the environment is a previous environment and the label creation functions are equivalent, then the new ep found in the label will be the same for both models, and environment access and hence variable sets will be preserved by the GOTO. The definition of procedure call in the original paper by Johnston is not suited for the definition which has been given here. Therefore, the definition from Johnston's later paper has been used [III.7]. The only problem is to be sure that the models find the same lexically enclosing contour for the procedure block. The models are equivalent if the ESEARCH procedures are equivalent, since both locate the nearest contour enclosing the procedure being invoked, with the exception of procedures which are passed as parameters to other procedures and then called. Equivalence is preserved if, as stated, the contour model environment search function traces back to the contour where the procedure was passed in an actual parameter list to begin its search. The definition of the ESEARCH and CALLX functions in the i-model were defined in exactly this way.

This completes the proof of equivalence of the contour model and the i-model.
THE VIENNA DEFINITION LANGUAGE

Ideally, this thesis would include a proof that the model which has been presented is equivalent to the Vienna Definition Language. This was the intent when the model was designed. However, the task of demonstrating an equivalence between something as conceptually simple as an i-model machine and something as complex as VDL is formidable. There simply is not space in this work for such a proof which would more properly be the subject of a separate work. An outline of a proof will be given, however.

An Outline of a Proof of Equivalence

There are basically two parts of VDL. The first is the object and its family of notation. The second is the Abstract Definition Machine. The exposition of the object in the literature is reasonably consistent, and equivalence in this area is not difficult to show. The VDL object is equivalent to the variable as defined in this thesis. An elementary object in VDL may be expressed in the notation developed here as a variable

(NAME, value)

where NAME is a selector name and value is a VDL elementary object. A composite object in VDL is equivalent to the
variable as has been defined in Chapter II, with the object part an unordered set of variables. For example, the VDL object illustrated in Figure I-2 may be written as

\[ A = \{ (A, \{(s_1, e_1), (s_2, \{(s_3, e_3), (s_4, e_4), (s_5, e_5)\}) \} \} \].

The functional notation of VDL is based primarily on existing concepts, for the most part. No equivalence proof per se is needed in this area; however, certain equivalent notations must be developed to express the same primitives as are expressed in VDL. Examples are the list notation, the search functions, and the mutation operator. The key notation, that of composition of selectors is in essence the same as the dot notation which has been used in this work to qualify variable names. In the example given above, element \( s_5 \) is selected in the two systems in the following ways:

\[ s_5.s_2(A), \text{ vs } A.s_2.s_5 \]

where it should be understood that the lower case "s" is used to conform to the original description, and not because it is a parameter.

The Abstract Definition Machine (DM) specifications vary among the authors using it [III.8, III.9, III.10]. The notation of Lee seems to be the most general, and is probably the one which should be used. In that event, an equivalent i-model machine could then be defined as

\[ P_{vd1} = (V_0, V, V^*, \sigma, F, H) \]

where \( V \) is the state variable set,
\[ V = (V, \{\text{ID, C, COPY, DATA}\}). \]

\(V\) would be equivalent to the DM state \(\xi\). \(V_0\) would be a representation of the initial state of \(\xi\) required for abstract program simulation. \(V^*\) would again be the operational closure of \(V\). The fetch function \(F\) would correspond to picking one of the instructions of the control tree for execution according to the non-deterministic algorithm used for the VDL DM. \(F\) would also presumably have to perform the parameter binding for parameterized instructions. The \(\sigma\)-function would be the state transition function, which would be somewhat simpler than that of a DM, since the \(F\)-function would possibly absorb some of the work.

One of the most formidable problems, and one which is as yet not fully resolved, is that of providing a structure for instructions in the equivalence machine. It is clear that the instruction format in the instruction definition part of the machine, \(\text{ID}\), would have to be like

\[ \text{INST}_k = (\text{INST}_k, \{\text{SUBARG, PLIST, PRED}\}), \]

where \(\text{SUBARG}\) is a variable containing the name of the subject argument formal parameter, \(\text{PLIST}\) is a list of the other formal parameter names, and \(\text{PRED}\) is the executable part of the instruction. The \(\text{PRED}\) variable is the variable requiring the most attention. This is due to the complex notation which may be used. The concept of the notation itself is simple, being based on the McCarthy LISP predi-
cates. However, the addition of parameter facilities and composite selectors and the separation of instructions into two separate classes, will require that the instruction be very carefully defined in the i-model. Notationally, the result must be as rigorous as the VDL notation.

A proof of equivalence would probably proceed in the following manner:

(1) Establish a set of variables whose values are to be monitored. This would include the DATA component of both systems, plus structure information from the control tree, C. In this feature is a great deal of similarity to the Contour Model, in that the invariant part of the ALGORITHM, the ID in this case, is not used. The structure of the control tree is indicative of language syntax, and thus is important; the contents of the DATA part of the state represent semantics, and must also be monitored.

(2) The proof would be an induction on the number of instructions executed by the twin machines, as has been the case with other models in this chapter.

(3) The basis for the equivalence proof would be the establishment of a unique (self-replacing) instruction in C when the systems are initialized to their respective initial states. Since the initial instruction is unique, the basis may be proved directly.

(4) The induction step would assume that the contents of the DATA parts of the states and the structure of the
control trees were the same through \( k \) steps. The non-determinacy of the fetch function of VDL would have to be dealt with at this point. VDL is defined such that the order in which instructions are executed is immaterial as long as they are terminal nodes. Thus, one may define an ordering for the execution of terminal nodes. For example,

(4.1) Start at the root node for \( C \), and follow the selector for the successor (s-succ(i)) which has the smallest index \( i \).

(4.2) Examine the instruction obtained. If it has no successors, then execute it. Otherwise, choose the successor with the smallest index and repeat 4.2.

(5) The results of the execution of instructions must be related among the two models after the equivalent instructions have been fetched. The two cases to consider are the self-replacing and the value-returning instructions. The proof would have to proceed thus:

(5.1) Case: self-replacing: Using the inductive hypothesis, the same instruction is executed by both models. Therefore, the resulting new instruction trees for the two models must have equivalent structures after the execution of the instructions. That is to say, the only part of the control tree which is modified is the deletion of the original instruction node and its replacement by equivalent structures in each model.

(5.2) Case: value-returning: If the two value-
returning instructions are equivalent, and if the parameter passing mechanism has been demonstrated to yield equivalent results (this would probably require an additional Lemma), then the same changes to the state (DATA) should be made. The effect on C of the execution of value-returning instructions is by definition the same: they are deleted from the control tree.

This would complete the proof.

Problems

The most difficult problems are those of instruction equivalence, especially as regards parameters. Lee mentions several pathological cases which must be considered [III.11]. These must be dealt with and shown to be equivalent for the two models, perhaps in a separate Lemma, as mentioned above. Further additional problems lie in some of the functions which VDL uses, notably the 1 and \( \tau \) functions which search for objects in sets which satisfy specified predicates. The \( \tau \)-function returns all such objects, while the 1-function returns the unique object satisfying the predicates. An equivalent notation must be defined in order to simulate a VDL process which uses these functions.

Advantages of I-Model Notation
There are some advantages which accrue to the use of the i-model notation. The first is clarity. Appendix A contains a formal description of the semantics of the language BASIC, in the notation of an i-model. This description is patterned after the same description by Lee [III.12] for the purpose of comparison; it is not necessarily the best specification for such an endeavor. The purpose is to give a flavor of the relative clarity of the i-model notation as opposed to the VDL notation for the same process.

The primary reasons for the clarity of the i-model notation relative to the VDL notation are the lack of the construct "s-" and the fact that variable qualification works from left to right, whereas composite selection works from right to left. The notation of VDL has frightened off many of its potential converts for just these reasons. The i-model notation, on the other hand, was designed with the user in mind. Further, the usage of both capital letters and lower case letters communicates additional information in the specification of instructions. When capital letters are used, a variable or component of a variable is known and concrete. When small letters are used, it means that parameters of an instruction are to be used, and that therefore they are of a dynamic nature. The reader could visualize where the parameters of instruction are being used. The construct
would indicate that the parameter of the instruction, "text", has a component which is a line number, a fact which would not be so easily seen in the VDL equivalent s-linenumber(text).

An additional advantage of the i-model notation is realizability. It may be possible to write an interpreter which would simulate the material in Appendix B, though no such claim is made at this time. This would be possible because of the clear distinction of parameters and a simplified functional notation which it should be possible to develop. The only foreseeable problem is the lack of a rigorous data description language. The reader will note that some of the data descriptions in Appendix B are rather informal.

THE MULTICS SYSTEM

The i-model system will now be applied to a system rather than the models to which it has been applied earlier in this Chapter. An i-model for address formation in Multics will be developed. A few remarks about the use of the same model at the system level will also be given, in order to give the reader a feel, however fleeting, for the applicability of the model at different levels of the same system.
The addressing i-model to be developed will be equivalent to non-paged address formation on the GE/Honeywell 645, which is the only computer on which Multics has been implemented. (The reason is the reverse of the usual situation. The system is primarily programmed in transportable PL/I [III.13]. No hardware exists to support the addressing needed, however.) The non-paged case is chosen for reasons of simplicity. It allows much detail to be discarded without significant loss of generality. The principal source for the description to follow is the work by Organick [III.14]; the reader may refer to it for much helpful additional description and many illustrative diagrams. The reader's forebearance is asked in the following lengthy discussion of Multics addressing. It is felt that the description must be included; those already familiar with the system may skip this section.

Multics Addressing

The bases of a Multics process are three: (1) The descriptor base register (dbr), (2) The process base register (pbr), and (3) The set of eight address base registers (abr). The dbr points to the core address of the descriptor segment for the process. This is the entire addressing space for the process. When a segment is first referenced, its name is searched in the directory system. If the name
is found, it is entered in a table unique to the process called the Known Segment Table. Its index in this table becomes its *segment number*. Notationally, segment "seg" is written <seg>; its segment number is written seg#. A descriptor for the segment as it is loaded into core is created and placed in the descriptor segment at the entry with index seg#. The format of the descriptor is given in Figure III-5. It may be seen from the diagram that the descriptor describes known segments as to type, and it indicates access privileges for the segments derived from the directory system. The descriptor also indicates the absence of data from core memory.

A reference to location j of segment <k> is written in the form <k>|[j]. Assume that k# is 2 and j is 5. The addressing using the dbr would occur as in Figure III-6.

The pbr holds the number S# of the segment currently being executed. The offset into the segment is supplied by the instruction counter ic. The addressing of the current instruction is derived from the two registers in combination.

It is not possible for an assembly language instruction of the form

\[
\text{op } k\#|j,6
\]

to be assembled as written. This is true because k# is not known until run-time. To link-edit the program would violate the principle of pure procedures in Multics. The
<table>
<thead>
<tr>
<th>ADDRESS (OF PAGE TABLE OR SEG.)</th>
<th>SIZE</th>
<th>DESCRIPTOR FIELD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17 18</td>
<td>26 27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE SIZE</th>
<th>SEG</th>
<th>N.U.</th>
<th>FAULT CODE IF NOT RES.</th>
<th>SEG. TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1024</td>
<td>1</td>
<td>NO</td>
<td>WR</td>
<td>MM=MASTER MODE</td>
</tr>
<tr>
<td>1-64</td>
<td></td>
<td></td>
<td>RD</td>
<td></td>
</tr>
</tbody>
</table>

0 MM 0 MM 0 MISSING (NOT RES)
WRITE READ 1 DATA
ALLOW ALLOW 2 PROCEDURE
Figure III-6.

Address Formation Using Segment Offset Pair \(<k>j\).
four abr pairs offer a means of solving the problem. Each even-odd pair of abrs contains a segment offset and a segment number, respectively. Pair 0-1 (the argument pair ap) is used to address the arguments of the current procedure. Pair 2-3 (the base pair bp) is used as a general pointer. Pair 4-5 (the link pair lp) is used to point to the linkage segment for the procedure (see below). Pair 6-7 (the stack pair sp) is used to point to a stack segment for the procedure.

Assuming that the segment k is addressed by the base pair the above instruction becomes

\[ \text{op } \text{bp}|j,6 \]

where bp is a designator for the base pair registers, 6 is the number of an index register, and j is the offset from the indexed value. The address formation process is illustrated in Figure III-7.

The other type of addressing that is of interest in this discussion is indirection via its (indirect-through-segment) pairs. These pairs of words in memory contain a segment number, an offset, an index register designator, and a further indirection bit. In other words, they offer the addressing features of an instruction and the abrs combined. Its pairs, whose structure is illustrated in Figure III-8, are vital to the run-time binding used in Multics. The segment number and offset are filled in dynamically when the segment is known. They are initialized
Figure III-7.
Address Formation Using Base Pair and Index Register.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>17</th>
<th>30</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>TAG</td>
<td>&quot;its&quot;</td>
</tr>
<tr>
<td>pointer offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>segment number</td>
<td></td>
<td></td>
<td>index reg</td>
<td>*</td>
</tr>
</tbody>
</table>

* indicates further indirect addressing.

Figure III-8.
An ITS Pair.
by the assemblers and compilers to symbolic references, which are used for searches at run time. When the information is found, it is converted into segment numbers and offsets. Assembled programs make most of their references through these pairs whose addresses relative to the programs are known to access data whose address cannot be known until run-time. An instruction of the form

\[ \text{op } \langle \text{seg} \rangle | \langle \text{loc} \rangle + i, j \]

where both "seg" and "loc" are symbolic and i is an integer offset from \( \langle \text{loc} \rangle \) the contents of index register j, is assembled as

\[ \text{op } \text{lpp} | k, j^* \]

The lpp is a designator for the link pair registers. The k is an offset into a linkage segment \( \langle \text{a.link} \rangle \) (where the instruction is part of segment \( \langle \text{a} \rangle \)). The * indicates indirect addressing through the its pair to be found at \( \langle \text{a.link} \rangle | k \). The formation of the actual address is illustrated in Figure III-9.

This completes a brief review of the significant features of Multics addressing.

An Equivalent I-Model for Multics Address Formation

The model for address formation in the Multics hardware presupposes the existences of a set of variables as follows:
Figure III-9.
Address Formation Using ITS Pairs.
(1) PROCESS - a composite variable whose index set I = 0 ≤ i ≤ max currently active process and whose components are:

(2) PROCESS.i - a composite variable = the dbr for process i. The index set for PROCESS.i is J = 0 ≤ j ≤ max seg # for process i, and whose components are:

(3) composite objects corresponding to Multics segment descriptor words. They carry the same information in the type code fields. The components of each of these are the Multics segments themselves.

(4) INDEXREG - a composite variable indexed from 0 to 7, each component of which is an 18-bit integer.

(5) BASEREG - a composite variable also indexed from 0 to 7, each component of which is a composite variable with the following structure:

   BASEREG.i = (BASEREG.i, {VAL, TAG, TYPE, LOCK})
   VAL = (VAL, an 18 bit integer)
   TAG = (TAG, a three bit integer)
   TYPE = (TYPE, a bit = 0 for pointer, 1 for base)
   LOCK = (LOCK, a bit = 0 if register may be user modified)

(6) An its pair is an object with 4 components which are variables:

   POINTER = (POINTER, 18 bit integer)
   INDEX = (INDEX, an 18 bit segment number)
   XREG = (XREG, a 3 bit index register number)
   STARBIT = (STARBIT, a bit = 1 for indirection)
(7) Additional auxiliary registers (variables):
   TBR = (TBR, 18 bits), the temporary base register
   TPR = (TPR, 18 bits), the temporary pointer reg.
   IC = (IC, 18 bits), the instruction counter
   PBR = (PBR, 18 bits), the procedure base register,
   which holds the segment number of the current
   process.

(8) INSTRUCTION - a 36 bit register which holds the
    instruction to be executed.

   INSTRUCTION=(INSTRUCTION,{TYPE,BASEREG,Y',Y,OP,STAR,
                        INDEXBIT,XREG})

   TYPE = (TYPE, bit =1 for instructions using base
           registers, 0 otherwise)

   BASEREG = (BASEREG, 3 bit base register designator)

   Y' = (Y', 15 bit signed integer literal displacement)

   Y = (Y,BASEREG,Y' used as the literal when
        TYPE=0)

   OP = (OP, 11 bit op code field)

   STAR = (STAR, bit =1 if indirection desired)

   INDEXBIT = (INDEXBIT, bit =1 if index register
               is to be added to resulting address)

   XREG = (XREG, 3 bit index register number used
           if INDEXBIT=1)

The above structures are illustrated in Figure III-10.

The Process machine is
Figure III-10.

The PROCESS and INSTRUCTION Variables for Multics.
\[ P_M = (V_0, V, V^*, \sigma, F, H) \]

\( V_0 \) is unspecified.

\( V \) is the set of variables described above.

\( F \) is defined as

\[
F(V) = (\text{INSTRUCTION}, \text{type}(\text{PROCESS}.i.(\text{PBR})) = \text{composite} \rightarrow \text{indirectype}(\text{PROCESS}.i.(\text{PBR})) = \text{code} \rightarrow \text{PROCESS}.i.(\text{PBR}).(\text{IC}))
\]

and is undefined if all the conditions do not hold. \( F \) is also undefined if the indexing operations using \( i, \text{PBR}, \) and \( \text{IC} \) are out of range.

The address formation being the only item under discussion, the state transition function is not the full state transition of the processor. This is one of the functional conveniences of the model, in that the process of address formation may be modelled separately from the full instruction set. A model for the instruction set could use the model developed here as a "black box" to obtain the data it needs to transform.

The state transition for address formation in general has several states due to the possibility of several type of indirection, and the use of address base registers, and/or its pairs. The cases of interest, however, require only two states. These are the use of address base registers and its pairs at the same time. This is the most powerful mode and it, more than any other, distinguishes
the Multics symbolic addressing system.

The initial state will be called \( E_0 \); state \( E_1 \) is an intermediate state. The transformations which occur in each are:

\[
\begin{align*}
E_0: & \quad TBR \leftarrow \text{BASEREGS.}(\text{INSTRUCTION.TAG}) \\
& \quad TPR \leftarrow \text{BASEREGS.}(\text{INSTRUCTION.TAG}+1)+\text{INSTRUCTION.Y}+ \\
& \quad \text{INDEXBIT} \times \text{INDEXREG.}(\text{INSTRUCTION.XREG}) \\
& \quad \text{INDIRECT} \leftarrow \text{INSTRUCTION.STAR}
\end{align*}
\]

then

\[
\begin{align*}
& \quad \text{OPERAND1} \leftarrow \text{PROCESS.i.}(TBR).(TPR) \\
& \quad \text{OPERAND2} \leftarrow \text{PROCESS.i.}(TBR).((TPR)+1)
\end{align*}
\]

(A simplification has been made here--\text{OPERAND2} is only fetched when indirection is specified and the address of the indirect word is even.)

If \( \text{INDIRECT} = 1 \), state \( E_1 \) is entered; otherwise, \( \sigma \) is done and the operand is in \( \text{OPERAND1} \).

State \( E_1 \) uses the \( \text{OPERAND1} \) and \( \text{OPERAND2} \) as an its pair and modifies the effective address in the following way:

\[
\begin{align*}
E_1: & \quad TBR \leftarrow \text{OPERAND1.Y} \quad \text{(where Y, INDEX, and STAR are as defined for INSTRUCTION)} \\
& \quad TPR \leftarrow \text{OPERAND2.Y}+ \\
& \quad \text{OPERAND2.}(\text{INDEXBIT} \times \text{INDEXREG.}(\text{OPERAND2.XREG})) \\
& \quad \text{INDIRECT} \leftarrow \text{OPERAND2.STAR}
\end{align*}
\]

then

\[
\begin{align*}
& \quad \text{OPERAND1} \leftarrow \text{PROCESS.i.}(TBR).(TPR) \\
& \quad \text{OPERAND2} \leftarrow \text{PROCESS.i.}(TBR).((TPR)+1)
\end{align*}
\]
If INDIRECT = 1 state E₁ is re-entered; otherwise, address formation is complete and the operand is in OPERAND1.

The System Level

When the addressing hardware in Multics attempts to reference a segment which is not in core, system action is required to bring the segment in from the file system or to create it, as appropriate. These macroscopic actions on the part of the system may be modelled as an i-model using much the same information as already given for the address formation model. The actual process by which the system would perform the file handling actions will not be modelled in detail in this work. A few comments about the modelling process are given.

When a segment fault occurs in the model which has been presented, the offending its pair is located in OPERAND1 and OPERAND2. The contents of OPERAND1 is an index into the currently executing segment, e.g. <a³>. The data in the segment is the symbolic representation of the segment name to be searched. OPERAND2 contains a similar pointer for the offset label expression. The action of the system would therefore have to be something like:

SAVPBR ← PBR
SAVEIC ← IC
SAVDBR ← PROCESS.i
SAVPBR = PBR
NAME = $\text{OPERAND1}$
OFFSETNAME = $\text{OPERAND2}$

Save other registers and set up the system process registers. Assume that the system process is process 0.

SEARCH(DIR\text{E}CTORY.$i$, PROCESS.$(\text{SAVDBR}).(\text{SAVPBR}).[\text{NAME}]$)

which looks in the segment which contains the entry for the symbolic name of the segment.

If not found and access is read, then fail.

If not found and access is write, create a segment named as PROCESS.$(\text{SAVDBR}).(\text{SAVPBR}).[\text{NAME}]$ and proceed.

If found, then perform a search in the symbol table for the found segment to locate the offset expression

PROCESS.$(\text{SAVDBR}).(\text{SAVPBR}).[\text{OFFSETNAME}]$.

Assign a segment number and place this in the next available entry of KST.$i$, where KST is a composite variable whose components are the Known Segment Tables for each process.

Fill the segment number and offset into the its pair and return.

Many of the same variables are used at the system level as are used at the hardware level, with the exception
of the auxiliary variables SAVDBR, etc., and the DIRECTORY, which is a composite variable whose components are variables which are Multics directories and segments.

The above highly informal discussion would not, of course, be the actual method employed by a system designer to model his system or the researcher to obtain a vehicle for proving that his system was e.g. deadlock free. However, the above discussion might be appropriate for an informative work or as the means of communication between members of a design team, such as a logic designer and a system programmer. The notation is flexible enough to be all of these things.
CHAPTER IV
CONCLUSIONS

The original objective of the research leading to this thesis was to pursue topics in the area of operating systems and resource allocation. It was found, however, that there has been a lack of a suitable notation in which to express problems and with which to obtain results. Therefore, it was decided to pursue the new goal of developing an easy to use, yet rigorous notation for the expression of computational processes. An examination of the results of this thesis indicates that the goal has been in large measure met, although there is still room for improvement.

The Basis for a Model

Preliminary examination of the computation problem revealed that there are in fact two aspects to a computation, those being the processor performing the computation, and the user of the processor who is expecting a result from the process. It was found that processors tend to deal at all levels, from hardware to operating system, with certain entities with common characteristics, whatever their outward structure. These entities have been given formal definition as objects. Similarly, users (whether
they be hardware designers, applications programmers, or operating systems designers) all tend to think in terms of named entities whose structure may or not accurately reflect the entities which processors manipulate. These common named entities were also given formal definition, as variables.

A model has been derived from these concepts which is applicable to the vast range of activities called computation, from the microcode level of a processor to the operating system resource allocation algorithm. The model is based on automata theory, which is the only area of mathematics capable of dealing with sequential processes. Modifications to the basic automata theory have been made to account for the definitions of the object and the variable. In particular, the memory of a process machine is defined to be a set of variables rather than a tape or stack. The variable, which is defined as a name/object or name/variable-set pair, is addressed in a random access symbolic manner instead of sequentially on a tape. The model is thus divorced from considerations of data management.

The definition of certain parts of the model are left to be defined in relation to the particular process to be modelled. These are the state transition function, $\sigma$, and an instruction fetch function, $F$. For a particular process, these two functions are specified, as is a set of
variables which is called the state variable set. It is possible, by judicious choice of variables and functions to model the same system, or different parts of a system simultaneously. This process was briefly outlined in Chapter III in the section on the Multics system. The address formation in the hardware was modelled fully. It was shown that to also model the related process of segment search in the directory structure would require but little change in the variable set for the addressing model. The implications of this fact are discussed below.

Results—Relation to Existing Models

The presentation of a new model for computational processes is usually greeted with skepticism, as there are a number of models already in existence. Therefore, some effort was expended to justify the new model, which has been termed an integrated, or i-model, in relation to some of the more widely known existing models. In particular, the ISP notation for hardware, the contour model for programming languages, and the Vienna Definition Language for general applications were examined. It was shown informally that the data structures of ISP could be duplicated in i-model variables. This result formed the heart of a proof that any concept which can be modelled in ISP may also be modelled in an i-model. The i-model system has a certain
formality which could conceivably allow an interpreter to accept a description of a processor in i-model notation and produce as a result a simulator for that processor. This could not be done with ISP, though the notation could be modified to be rigorous enough for such an endeavor.

The Contour Model for block structured languages as outlined briefly in Chapter I has gained recognition for its treatment of the problems which arise in block structured languages, most notably the problems of changing name/object mappings as a result of block entry, procedure calls, and non-sequential transfers of control. The strength of the Contour Model is pedagogical—it has the ability to graphically display the environments encountered by processors as they perform these actions. The Contour Model also has a formalism for describing the effects of instructions which cause environments to be changed. A proof has been presented in Chapter III that an i-model may be found which is equivalent to this formalism.

The Vienna Definition Language has been used to model the semantics of programming languages, most notably PL/I and BASIC; it has also been used to model algorithms and processors, much in the same spirit as the i-model which has been presented here. The formalisms of VDL, however, while powerful, seem divorced from the terms in which potential users of it might think. Potential users are
frightened away by the forbidding set of data and instruction definitions which must be mastered in order to use VDL constructively. Therefore, it would be desirable if the i-model notation, which seems to be quite natural in relation to VDL, could be shown to be equivalent to it. It was decided that a proof of such a result would be outside the scope of this work. However, an outline of a proof is given. Further, a formal definition of the language BASIC is given (in Appendix A) which is parallel to that given by Lee. Examination of the two definitions shows the relative clarity of the i-model notation.

Further Work

The i-model system is not without shortcomings and faults. Among these are the lack of a precise notation for describing objects and variables. The notation used in Appendix A is the most formal yet devised for the i-model, and it is rather weak in spots. This is a problem which should disappear as experience is gained with the system. It is only a few man-months old at this point as compared to the several man-years of the models for which equivalence was proved in Chapter III. Therefore, the first items on a list of further steps for the i-model would be to formalize the data description and state transition notations.

The proof of equivalence with the Vienna Definition
Language should be possible when the data description problem is solved. The proof would probably be a suitable subject for a paper or perhaps a master's thesis, if combined with a formalization of the data description facilities. The claim was made in Chapter III that an interpreter could be written to accept i-model notation similar to VDL notation to produce the effect of the execution of the VDL Abstract Definition Machine. An attempt should be made to write such an interpreter.

An interesting possibility would be a combined hardware-programming language interpreter which would produce machine code for a machine given the machine specification and a language specification. Though admittedly wild speculation, it might be instructive to attempt such an interpreter.

Other additional work would fall in the area of applications of the model. One such area of particular value would be a hardware description of the R-2 computer system. The R-2 is based on many of the principles used in the design of the i-model system, and it should therefore be a rather simple task to model it in i-model notation. The attempt should be made to model several levels of the system in the hope that the use of but one model for all levels of the system would result in a more coherent overall system design.
REFERENCES FOR CHAPTER I


[I.6] Ibid.


[I.9] Iliffe, op. cit.


[I.11] The case in point is the FORTRAN LOAD AND GO (FLAG) Compiler for the IBM 1620 written and used at Rice University from 1964 to 1970.


[I.16] Bell and Newell, p. 28.


REFERENCES FOR CHAPTER II


[II.4] Lucas and Walk op. cit. [I.17].

1 The type code may be implicit if the nature of all objects is known at all times. This is usually the case in e.g. BASIC.


2 The MU5, under development at Manchester University has a name store which apparently will implement this system in some sense. See R. N. Ibbett, "The MU5 Instruction Pipeline", Computer Journal 15,1 (Feb. 1972) pp. 43-51.


REFERENCES FOR CHAPTER III

[III.1] Bell and Newell, op. cit. [I.14], p.23.


[III.6] Ibid., §2.5.

[III.7] Ibid., pp. II.29-II.33.

[III.8] Lucas and Walk, op cit., [I.20].


[III.11] Lee, pp. 94-104.


APPENDIX A

A DESCRIPTION OF THE LANGUAGE BASIC

What follows is Lee's description in VDL of the language BASIC as translated into the notation of an i-model machine. The abstract syntax will be given first. This corresponds to a specification of the set of variables $V$, for an i-model machine. This description will be followed by a description of an interpreter for the machine, that is, the state transition function.

Abstract Syntax

$DATA = (DATA, \{TEXT, STG, INPUT, ATTR, OUTPUT, FORSTACK, FUNCTIONS\})$

$TEXT = (TEXT, \{line\})$

where line is in the set of line numbers of the program.

Line is used as a variable qualifier and not as an index because line numbers need not be consecutive.

$TEXT.line = (TEXT.line, \{LINE, STNAME, *\})$

Where * is defined as

$TEXT.line.STNAME = REM \rightarrow \Omega$

$TEXT.line.STNAME = END \rightarrow \Omega$

$TEXT.line.STNAME = STOP \rightarrow \Omega$

$TEXT.line.STNAME = GOSUB \rightarrow (\text{DESTINATION, linenum})$

$TEXT.line.STNAME = RETURN \rightarrow \Omega$
TEXT.line.STNAME=FOR → (INDEX, var), (INITIAL, expr),
   (LIMIT, expr), (INCREMENT, expr)
TEXT.line.STNAME=NEXT → (INDEX, var)
TEXT.line.STNAME=LET → (VARIABLE, var), (EXPR, expr)
TEXT.line.STNAME=READ → (LOAD, var list)
TEXT.line.STNAME=RESTORE → Ω
TEXT.line.STNAME=PRINT → (OUTPUT, expr list)
TEXT.line.STNAME=IF → (BOOLEAN, {
   (RELATION, relationalchar),
   (EXP1, expr), (EXP2, expr)}) (DESTINATION, linenumber)
TEXT.line.STNAME=FNEND → (FNAME, fname)
TEXT.line.STNAME=ON → (EXP, expr), (LINELIST, linenumber list)
INPUT=(INPUT, {(KEY, integer), (DATA, externalrep list)})
ATTR=(ATTR, {var})
ATTR.var=(ATTR.var, {(TYPE, typecode),
   (ELEMENT, *)})

Where * is defined as
TYPE=simple → location
TYPE=function → location
TYPE=vector → (UPBD, integer), (ELEMENT, location)
TYPE=table → (UPBD, integer), (ELEMENT,
   {(UPBD, integer), (ELEMENT, location)})
OUTPUT=(OUTPUT, val list)
where val=(val, {(TYPE, otype), (VAL, *)})
where * is defined as
TYPE=externalrep → external representation of
a value

TYPE=literal → char list

TYPE=punct → character

TYPE='CR' → carriage-return, line feed

FUNCTIONS=(FUNCTIONS, functdef)

functdef=(functdef, {(PAR, integer list), (EXP, {
   ((TYPE, ftype), (VAL, *))})})

Where * is defined as

TYPE=expr → expr

TYPE=defblock → line number group

location = integer

otype = externalrep or literal or punct or 'CR'

var=simple or vector or function or table

expr=infixexpr or prefixexpr

infixexpr=(infixexpr, {(OPERAND1, expr),
   (OPERAND2, expr), (OPERATOR, optr)})

prefixexpr=(prefixexpr,
   {(OPERAND, expr), (OPERATOR, prefixoptr)})

optr='+' or '-' or '*' or '/' or '+'

prefixoptr='+' or '-'

relationalchar='=' or '<' or '>' or '<=' or '>=' or '<>'

punct=',' or ';

literal = char list

if a variable is used with the suffix "list", the variable type is assumed to be in the form of
a list structure, with system functions
head(list), tail(list) and elem(i,list) defined
in the obvious way.

This completes the specification of the abstract syntax
for BASIC.

The State Transition and Sequencing Functions

Order of execution: As in VDL, indented instructions are
executed first, and in parallel with those in the same
group of indented functions.
Sequencing: The same rule as VDL is used. Terminal nodes,
i.e. those furthest indented in the description to follow,
are executed before others in a non-deterministic way.
With respect to notation: the use of the ";" implies that
the following statements should be indented. The use of
the "," separates inner statements of a group.
Initial State: The initial state of the interpreter is
given as

\[ V_0 : V,C=\text{execute-program(DATA,TEXT)} \]

Instruction Definitions: The instructions of the machine
are now given:

(1) \text{execute-program(text)} =
    \text{execute-statement(next-line-no(text,0),TEXT,\emptyset)}

(2) \text{execute-statement(lineno,text,stack)} =
lineno=Ω or text.lineno=Ω → error

   text.lineno.STNAME=Ω or

   text.lineno.STNAME=REM →

       execute-statement(next-line-no(text,lineno),
                        text,stack)

   text.lineno.STNAME=END or

   text.lineno.STNAME=STOP → null

   text.lineno.STNAME=GOTO →

       execute-statement(text.lineno.DESTINATION,text,stack)

   text.lineno.STNAME=GOSUB →

       execute-statement(text.lineno.DESTINATION,text,
                         push(next-line-no(text,lineno),stack)

   text.lineno.STNAME=RETURN →

       execute-statement(top(stack),text,pop(stack))

   text.lineno.STNAME=FOR →

       execute-for-st(text.lineno,text,stack)

   text.lineno.STNAME=NEXT →

       execute-next-st(DATA.FORSTACK;text,lineno,stack)

   text.lineno.STNAME=LET →

       store(DATA.STG;loc,value);

       loc:get-location(text.lineno.VARIABLE)

       value:evaluate-expr(text.lineno.EXPR)

   text.lineno.STNAME=READ →

       execute-read-st(DATA.STG;lineno,DATA.INPUT,text,stack)

   text.lineno.STNAME=RESTORE →

       restore-key(DATA.INPUT.KEY;)

execute-statement(c,text,stack)
  c:compare-relation(a,b,text.lineno.BOOLEAN.RELATION,
  text.lineno.DESTINATION,next-line-no(text,lineno));
  a:evaluate-expr(text.lineno.BOOLEAN.EXP1),
  b:evaluate-expr(text.lineno.BOOLEAN.EXP2)
pass(result)
  result:fetch(DATA.STG,loc)
  loc:get-location(text.lineno.FNAME)
text.lineno.STNAME=ON +
  int-on-st(a,text.lineno.LINELIST,text,stack);
  a:evaluate-expr(text.lineno.EXP)

(3) execute-for-st(for-st,text,stack)=
  search(DATA.FORSTACK,for-st.INDEX)≠∅ + error
  T + execute-statement(a,text,stack);
  a:compare(DATA.FORSTACK;init,
    least(those-next(for-st.INDEX,for-st.LINE),
    text));
  store(DATA.STG;loc,init),
  stackx(DATA.FORSTACK;for-st.LINE,for-ST.INDEX,
     limit,inc);
  limit:evaluate-expr(for-st.LIMIT),
  init :evaluate-expr(for-st.INITIAL),
  inc :evaluate-expr(for-st.INCREMENT)
(4) \text{stackx(for-stack;lineno,variable,limit,inc) =}
\begin{align*}
\text{TEMP.LINE} & \leftarrow \text{lineno}, \\
\text{TEMP.INDEX} & \leftarrow \text{variable}, \\
\text{TEMP.LIMIT} & \leftarrow \text{limit}, \\
\text{TEMP.INC} & \leftarrow \text{inc}, \\
\text{push(TEMP,for-stack)} & 
\end{align*}

(5) \text{execute-next-st(for-stack;text,lineno,stack) =}
\begin{align*}
\text{forstack=} & \Omega + \text{error} \\
\text{text,lineno,INDEX} & \leftarrow \text{top(for-stack).INDEX} + \\
\text{execute-next-st(tail(for-stack);text,lineno,stack)} & \Rightarrow \\
T & \leftarrow \text{execute-statement(b;text,stack)};  \\
\text{b:compare(for-stack;a,lineno)}; \\
\text{a:increment(for-stack)} & 
\end{align*}

(6) \text{increment(for-stack) =}
\begin{align*}
\text{pass(x);} \\
\text{store(DATA.STG;loc,next);} \\
\text{x:int-infix-expr(head(for-stack).INCREMENT,} \\
\text{value,'+');} \\
\text{value:fetch(DATA.STG,loc);} \\
\text{loc:get-location(head(for-stack).INDEX)} & 
\end{align*}

(7) \text{compare(for-stack,value,lineno) =}
\begin{align*}
\text{signum(head(for-stack).INCREMENT} \times \text{value} & 
\end{align*}
-head(for-stack).INIT)>0 +
PASS:next-line-no(DATA.TEXT,lineno)
   for-stack< tail(for-stack)
   T +
   PASS:next-line-no(DATA.TEXT,head(for-stack).LINE)

(8) fetch(stg,loc)=
    PASS:elem(loc, stg)

(9) store(stg;loc,value)=
    stg.(elem(loc)) ← value

(10) get-location(variable)=
    variable.TYPE=simple +
    get-loc-1(DATA.ATTR.variable.NAME)
    T +
    get-loc-2(DATA.ATTR.variable.NAME,variable)

(11) get-loc-1(symtab)=
    PASS:symtab:ELEMENT

(12) get-loc-2(symtab,variable)=
    symtab.ELEMENT.TYPE=elementary +
    variable.SUBSCR2=Ω + error
    T + map-3(symtab.UPBD,symtab.ELEMENT,subscript);
       subscript:evaluate-expr(variable.SUBSCR1)
T \rightarrow \text{variable.SUBSCR2}=\Omega \rightarrow \text{error}
T \rightarrow \text{map-4(symtab.UPBD, symtab.ELEMENT.UPBD, symtab.ELEMENT.ELEMENT, subscr1, subscr2)};
subscr1: \text{evaluate-expr(variable.SUBSCR1)},
subscr2: \text{evaluate-expr(variable.SUBSCR2)}

(13) \text{map-3(dim-1, dim-2, loc, subscript)} = 
\text{PASS: map-1(dim-1, loc, subscript)}

(14) \text{map-4(dim-1, dim-2, loc, subscr1, subscr2)} = 
\text{PASS: map-2(dim-1, dim-2, loc, subscr1, subscr2)}

(15) \text{evaluate-expr(exp)} =
\text{exp.TYPE=infix} \rightarrow
\text{int-infix-expr(a, b, exp.OPERATOR)};
\text{a: evaluate-expr(exp.OPERAND1)},
\text{b: evaluate-expr(exp.OPERAND2)}
\text{exp.TYPE=prefix} \rightarrow
\text{int-prefix-expr(a, exp.OPERATOR)};
\text{a: evaluate-expr(exp.OPERAND)}
\text{exp.TYPE=variable} \rightarrow
\text{fetch(DATA.STG, loc)};
\text{exp.TYPE=constant} \rightarrow
\text{pass(internal-rep(exp))}
\text{exp.TYPE=function-ref} \rightarrow
\text{int-function-ref(expression)}
pass(ele1(exp::NAME::VAR,
   DATA::FUNCTIONS.[exp::NAME::DEF].PAR))

(16) int-infix-expr(op1, op2, opr) =
   opr='+'  → PASS: op1 + op2
   opr='-'  → PASS: op1 - op2
   opr='*'  → PASS: op1 × op2
   opr='/'  → PASS: op1 ÷ op2
   opr='\'  → PASS: *op2 × op1

(17) int-prefix-expr(opd, opr) =
   opr='+'  → PASS: opd
   opr='-'  → PASS: - opd

(18) compare-relation(exp1, exp2, rel, dest, nextline) =
   rel='='  & (exp1=exp2) or
   rel='<'  & (exp1<exp2) or
   rel='>'  & (exp1>exp2) or
   rel='\=' & (exp1≤exp2) or
   rel='\=' & (exp1≥exp2) or
   rel='\>' & (exp1>exp2) → PASS: dest
   T → PASS: nextline

(19) int-on-st(value, list, text, stack) =
    elem(compress(value, length(list)), list), text, stack)
(20) int-function-ref(exp) =
   exp.NAME.TYPE=library & length(exp.ARG)=1 →
   int-library-fn(exp.NAME, x);
   x: evaluate-expr(head(exp.ARG))
   exp.NAME.TYPE=user →
   int-fn-def(DATA.FUNCTIONS.[exp.NAME].DEF);
   evaluate-args(DATA.FUNCTIONS.[exp.NAME].PAR; exp.ARG, 1)

(21) int-library-fn(name, arg) =
   pass(name(arg))

(22) int-fn-def(block) =
    block.TYPE=expr → evaluate-expr(block)
    block.TYPE=defblock → evaluate-program(block)

(23) evaluate-args(plist, arglist, n) =
    n > length(arg-list) → null
    T → evaluate-args(plist, arglist, n+1);
    store-par(elem(n, plist); a);
    a: evaluate-expr(elem(n, arglist))

(24) store-par(par; a)
    par + a

(25) execute-read-st(stg; lineno, input, text, stack) =
    execute-statement(next-line-no(text, lineno),
text, stack);

read(stg;input,text.lineno.LINE,1)

(26) read(stg;input,read-st,n)=
n>length(read-stLOAD) → null
T → read(stg;input,read-st,n+1);
store(loc,value);
loc :get-location(elem(n,read-stLOAD)),
value:get-data-val(DATA.INPUT.KEY;INPUT.DATA)

(27) get-data-val(key;data-list)=
key>length(data-list) → error
T → PASS:internal-rep(elem(key,data-list))
key+key+1

(28) restore-key(key;)=
key+1

(29) execute-print-st(lineno,text,stack)=
execute-statement(next-line-no(text,lineno),text,stack);
print(DATA.OUTPUT;text.lineno.OUTPUT,1)

(30) print(output;print-st,n)=
n>length(print-st) → add-to-output(output;'CR')
n=length(print-st)&elem(n,print-st).OTYPE=punct →
add-to-output(output;print-st,n+1)
elem(n,print-st).OTYPE=expr →
add-to-output(output;a);
        a:evaluate-expr(elem(n,print-st))
        T → print(output;elem(n,print-st))

(31) add-to-output(output;element)=
        is-int-value(element) → output+concatenate
        (output,external-rep(element))
        T → output+concatenate(output;element)

(32) least(list)=
        is-null-list(list) → error
        length(list)=1 → head(list)
        head(list)<head(tail(list))→
        least(concatenate(head(list),tail(tail(list))))
        T → least(tail(list))

(33) those-next(index,lineno,text)=
        makelist((τi)(text.i.STNAME)=NEXT &
        text.i.INDEX=index & i < lineno))

(34) compress(value,list-length)=
        value<1 → 1
        value>list-length → list-length
        T → value

(35) next-line-no(text,i)=

i>(i,j)(text.j.STNAME=END) + error
text.i=\Omega + next-line-no(text,i+1)
T + i