Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.)
North-Holland Publishing Company, (1978)

MODELS OF HIERARCHICAL MACHINE SUPPORT

D. M. Berry[%†*#]
Computer Science Department
University of California
Los Angeles, California 90024

M. Erlinger[† @]
Computer Science Department
University of California
Los Angeles, California 90024
Hughes Aircraft Co.
El Segundo, California 90230

J. B. Johnston[&†]
Computer Science Department
New Mexico State University
Las Cruces, New Mexico 88003

A. von Staa[§]
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro, Brasil

In a multilevel hierarchically designed and implemented operating system, there are several methods for supporting the abstract machine at any level: interpretation, enmasterization, virtualization, compilation, and software extension. This paper presents examples of each method and describes a framework for characterizing any method as a pair of information structure models and mappings between them. The framework is used to describe three of the methods and their examples (descriptions of the other methods may be found in [BEJS77]). The paper concludes with a view of an entire system as a possible multipeaked tower of such models.

1. Introduction

There is much talk about building a system hierarchically, bottom-up from a raw machine [Dij68, Bri70, ZR68, Bau73, Den73, Goo73]. Each level implements a higher level machine in terms of the primitives offered at that level. A typical system is shown in Figure 1.1.

[%]This research was supported in part by the National Science Foundation, Grant No. DCR74-08659.

[†]This research was supported in part by the U.S. Energy Research and Development Administration, Contract No. EY-76-S-03-0034, PA 214 (formerly E(04-3)-34, PA 214).

[*]This research was supported in part by IBM do Brasil.

[#]This research was supported in part by the National Science Foundation, Grant No. OIP 73-07346 A02.

[§]This research was supported in part by the CNPq (BRASIL) - NSF (USA) interchange program.

[&]This research was supported in part by the National Science Foundation, Grant No. MCS 75-15997.

[@]This research was supported in part by Hughes Aircraft Co.'s Ph.D. Fellowship.

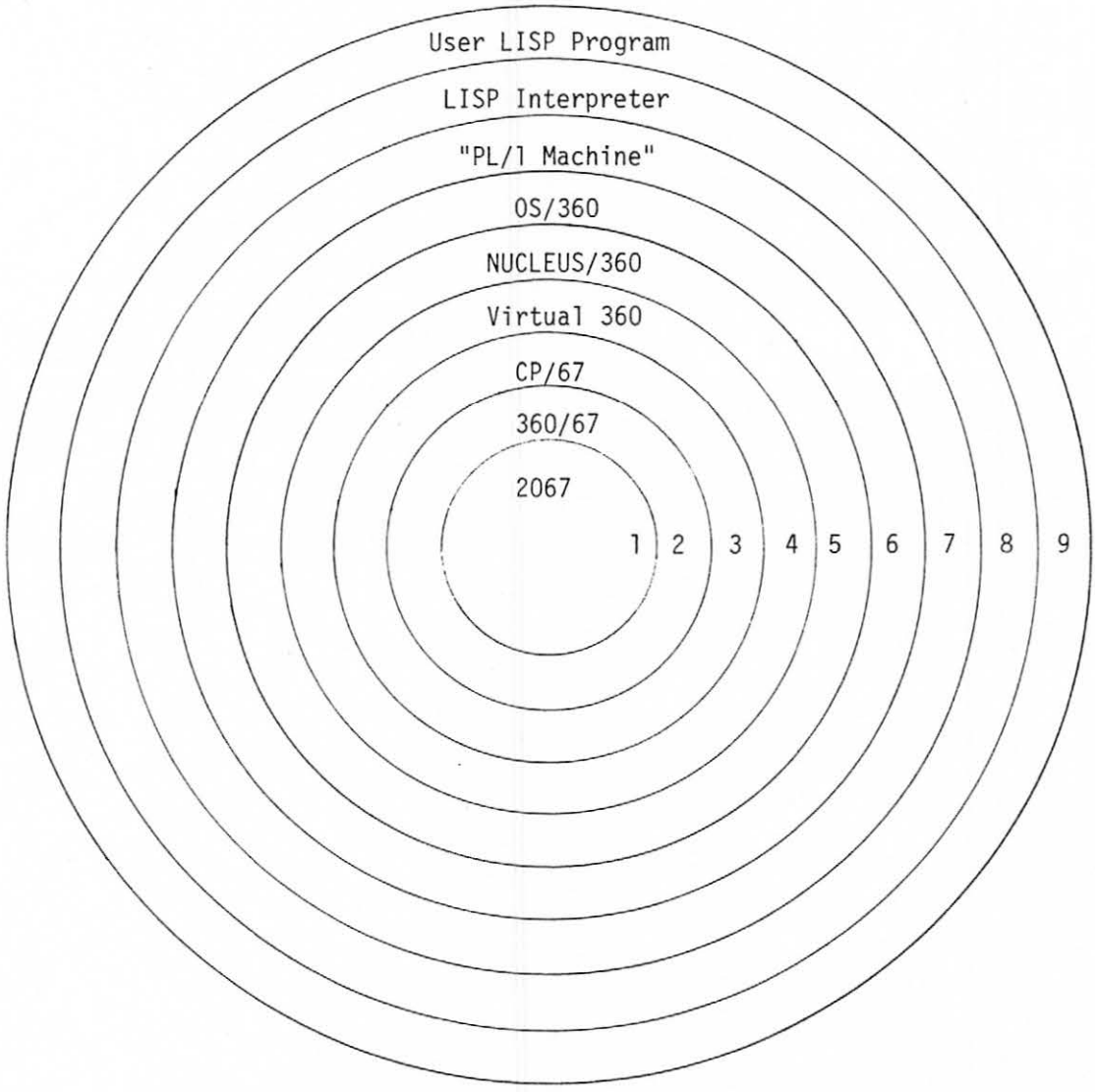


Figure 1.1

The machines from the lowest level on up are:

1. The raw IBM 2067 microprogrammable host machine
2. A system 360 model 67 microprogrammed on the 2067
3. A virtual machine monitor CP 67 which is the 360/67 extended with a number of routines for managing virtual machine
4. A virtual 360, supported by CP 67 which looks like a traditional 360
5. NUCLEUS/360 providing the basic supervisor routines which use the pri privileged instructions
6. Operating System/360 (OS/360) which is NUCLEUS/360 extended by a number of useful packages, compilers, utility routines and access methods
7. A PL/1 machine supported by compilation of PL/1 programs into 360 machine code combined with OS supervisor calls (i.e.,"OS machine" instructions)
8. A LISP interpreter written in PL/1
9. A user's programs written in LISP

Even cursory examination of the multilayered system shows that the progression from layer to layer is not done in a uniform manner. Five different methods can be identified.

1) Interpretation - The simulation of a computation at one level by direct manipulation of the data structures for that level by a program written in the language of the next lower level. If the lower level is a microprogrammable machine, interpretation is often called emulation. The steps from levels 1 to 2 and 7 to 8 are the examples of interpretation and the former is the one example of emulation.

2) Enmasterization - Extension of a machine not capable of performing certain privileged instructions by use of a non-privileged supervisor call instruction to cause trapping to supervisor state in which the privileged instructions may be performed. The jumps from levels 2 to 3 and from 4 to 5 are the cases of enmasterization in out example.

3) Virtualization - Support of a machine nearly identical to the supporting machine with nearly the same execution speed as the supporting machine. It requires that all sensitive instructions directly dealing with a resource be privileged and it usually uses enmasterization to have the supporting machine do the privileged instructions on behalf of the supported machine. The step from levels 3 to 4 is the example of virtualization.

4) Compilation - Translation of a source program in the language of the implemented machine to an object program in the language of the implementing machine whose effect as seen by the user is the same as the source program. The progression from levels 6 to 7 is the example of compilation in our hierarchy.

5) Software extension - The "extension" of a language capable of calling procedures in the language by a set of procedures written in the language. The language is made to appear more powerful in that at least the user of the procedures does not have to write them. This method of support is used in moving from levels 5 to 6.

If the input of the user program is considered a language then the step from levels 8 to 9 may be considered an example of interpretation.

This paper is a shortened version of a much larger work, [BEJS77], which attempts to clarify the differences between these methods of machine support. In [BEJS77], each method as described in terms of mappings between two Information Structure Models (ISMs) [Weg71], and each ISM is a variant of the Contour Model [Joh71].

In this paper we first define ISMs and ISMs for programming languages and then describe a portion of the CM sufficient for our purposes. The concepts of the Implemented and the Implementing Machines are offered, and we give a framework for our method descriptions by characterizing what a support method really is in terms of machines and mappings between machines. Then two of the methods are described in terms of these characterizing mappings. For descriptions of the other methods see [BEJS77].

The impetus for writing this work came from our attempt to understand "Nested Interpreters and System Structure" by Michael J. Manthey [Man75]. This report describes a contour model of multilevel interpretation. We have taken a different approach to modeling this particular phenomenon and have extended the approach to other and mixed methods of support.

2. Nondeterministic Information Structure Models

We will define a machine, real or abstract, by giving a nondeterministic information structure model (NDISM) that behaves like the machine. A computation of a program in the language of the machine will be described as a sequence of snapshots (instantaneous descriptions, core dumps) taken between successive instruction executions. Each computation starts off with an initial snapshot S_0 and proceeds through successive snapshots $S_1, S_2, \ldots$. Each snapshot is obtained from the previous by execution of some instruction, that is, by the application of

some transformation. Since some of the machines in [BEJS77] have multiple processors, it may be nondeterministic as to which instruction is executed next. Consequently the transformation is nondeterministic; that is, it maps a snapshot to a set of snapshots.

Therefore, we define a NDISM as a three tuple, (I,I_0,F), where I is a countable set of all possible snapshots, I_0 is the subset of I which is the set of all possible initial snapshots, and F is a transformation which maps a snapshot to a set of snapshots.

Definition 1. $M = (I,I_0,F)$ is a nondeterministic information structure model (NDISM) if and only if

1) I is a countable set of objects called snapshots.
2) $I_0 \subset I$ is the set of objects called initial snapshots.
3) F is a transformation of the form $F: I \rightarrow P(I)$, where P(I) denotes the set all subsets of I.

The transformation is applicable to a given snapshot if the transformation maps the snapshot to a non-null set of snapshots. A snapshot is transformable if the transformation is applicable to it; otherwise it is intransformable.

Definition 2. Let $M = (I,I_0,F)$ be an NDISM; let $S \in I$ be a snapshot. Then,

1) F is applicable to S if and only if $F(S) \neq \emptyset$.
2) S is transformable if and only if F is applicable to S.
3) S is intransformable if and only if S is not transformable.

A computation is a sequence of snapshots satisfying certain initial and inductive conditions; i.e., if the sequence is non-empty, then the first snapshot in the sequence, S_0, is an element of I_0; and, for all S_i in the sequence, $S_i \in F(S_{i-1})$.

However, this is not enough. Suppose $\langle S_0,S_1,S_2,S_3\rangle$ is a computation. Then clearly the sequences $\langle S_0\rangle$, $\langle S_0,S_1\rangle$ and $\langle S_0,S_1,S_2\rangle$ all satisfy the initial and inductive conditions and thus appear to be computations even though they are all "incomplete" subsequences of a computation. To fix this hole in the definition we add the stipulation that a computation is a sequence that is also not a proper initial subsequence of any other sequence satisfying the initial and inductive conditions.

Definition 3. Let $M = (I,I_0,F)$ be an NDISM. Then the sequence $C = \langle S_0,S_1,\ldots,S_i,\ldots\rangle$ is a computation in M if and only if

1) for all S_i in C, $S_i \in I$,
2) if $C \neq \langle\rangle$ (the empty sequence), then $S_0 \in I_0$,
3) for all S_i in C with $i > 0$, $S_i \in F(S_{i-1})$, and
4) for all sequences D satisfying 1), 2) and 3) above, C is not a proper initial subsequence of D.

We say that C is a computation of S_0 in M if the first snapshot of C is S_0. Also, for an NDISM M, we define the function M to give all computations in M of a given S_0.

Definition 4. Let $M = (I,I_0,F)$ be an NDISM; let $S_0 \in I_0$. Then

1) C is a computation of S_0 in M if and only if
   a) $C \neq <>$ is a computation in M.
   b) The first snapshot in C is S_0.
2) $M(S_0) = \{C | C \text{ is a computation of } S_0 \text{ in } M\}$.

Since every snapshot of a computation C except the first is obtained by transforming the previous, it is clear that there is at most one intransformable snapshot in C (there may be none, if C does not halt) and if there is an intransformable snapshot in C, it must be the last one.

THEOREM 1. Let $M = (I, I_0, F)$ be an NDISM; let C be a computation in M. Then

1) There exists at most one snapshot $S \in I$ in C such that S is intransformable.
2) If $S \in I$ is a snapshot in C such that S is intransformable, then is the last snapshot in C.
3) If $S \in I$ is the last snapshot in C, then S is intransformable.

We now have the right to speak of the unique intransformable snapshot S of a computation if such a snapshot exists. We call a computation that has one a halting computation and we call the intransformable snapshot the final snapshot of the computation.

Definition 5. Let $M = (I, I_0, F)$ be an NDISM; let C be a computation in M. Then

1) C halts if and only if for some $S \in I$, S is in C and S is intransformable,
2) final(C) is defined if and only if C halts,
3) final(C) = S if and only if S is in C and S is intransformable.

So far, NDISMs and computations have been defined independently of machines and their languages. Since the NDISMs given in the sequel are for modeling machines and the execution of programs in their languages, we must add assumptions under which an NDISM will be considered a model of a machine MACH with machine language L. In most machines and languages, we have input and output capabilities. Therefore, we assume that for each NDISM, M, modeling a machine MACH with language L, there are snapshot component selection functions called $\underline{input}_M$ and $\underline{output}_M$ which select the input and output lists of a snapshot. We also assume that there is a set of input lists, INPUT, and a set of output lists, OUTPUT, in which all possible input lists and output lists of integers, reals, booleans and character strings may be found.

Furthermore, we assume that there exists an initiation function $\underline{init}_M$, which produces an initial snapshot of M from a given program p in L and a given δ in INPUT. The $\underline{input}_M$ and $\underline{init}_M$ functions are assumed to be related in the following manner: the $\underline{input}_M$ of an initial snapshot S_0 is δ if and only if for some program p in L, S_0 is the result of initiation with p and δ.

Assumption 1. Let $M = (I, I_0, F)$ be an NDISM for a machine MACH with language L. Then

1) there exist countable sets INPUT and OUTPUT of lists of integers, reals, booleans and character strings. The empty list, <>, is in both INPUT and OUTPUT.
2) there exist functions, $input_M: I \rightarrow INPUT$ and $output_M: I \rightarrow OUTPUT$.
3) there exists a function, $init_M: (L \times INPUT) \rightarrow I_0$.
4) for all $S_0 \in I_0$, $input_M(S_0) = \delta$ if and only if for some $p \in L$, $init_M(p, \delta) = S_0$.

Therefore, the set of computations in M of the program p in L with input δ is denoted by $M(init_M(p,\delta))$.

Since we are concerned only with NDISMs for machines, we shall refer to them simply as NDISMs in the discussions that follow.

In this paper, all of our examples involve machines with one processor; thus, there will be at most one successor to a given snapshot. An NDISM for which this is true is called a deterministic ISM (DISM).

Definition 6. Let $M = (I, I_0, F)$ be an NDISM. Then

M is a deterministic information structure model (DISM) if and only if for all $S \in I$,
cardinality $(F(S)) \leq 1$ ∎

If we restrict our attention to DISMs, then a number of notational simplifications are possible.

Notation 1. Let $M = (I, I_0, F)$ be a DISM. Then the notation in the left column shall be used in place of the corresponding notation in the right column:

$F(S_i) = S_{i+1}$	$S_{i+1} \in F(S_i)$
$F(S_i)$ is undefined	$F(S_i) = \emptyset$
$M(S_0)$	the one element of $M(S_0)$ ∎

3. Contour Model

The contour model, introduced by Johnston [Joh69a, b, 71] is an NDISM which, because of the pictorial nature of its snapshots, has proved to be particularly suited for describing a variety of computational phenomena. These include nested declaration programming languages [Bry74a, OFP78], machines [Org73], and Multics-like systems [Joh75]. For a more complete description of its pedagogic use see [Bry74b].

The model we use here is the basic model enhanced by modification suggested in [CDMPS73, JBM74]. Because most operating systems are written in languages with compile or link edit time binding of nonlocal identifiers, i.e., those of the Algol family, we will restrict ourselves to considering the static-binding version of the contour model suitable for modeling the Algol family. In the terminology of [Joh73], we will use the STATIC complete identifier binding strategy. This version has been defined in VDL [Bry75].

3.1 Snapshots

In the contour model (CM), a snapshot consists of a time-invariant algorithm and a time-varying record of execution.

The algorithm consists of a sequence of instructions embedded in a nest of algorithm contours. See figure 3.2 for an algorithm corresponding to the source program of figure 3.1. Each algorithm contour corresponds to a block or procedure. The instructions or contours nested inside contour a A correspond to statements or blocks or procedures nested inside the block or procedure corresponding to contour A.

An algorithm contour has in its upper left hand corner a declaration array with one subcell for each identifier declared in the corresponding block or procedure; in the subcell lies the identifier paired with its type. If an algorithm contour is that of a procedure its declaration array will have as its first subcell, an entry explicitly declaring a variable to hold the return label of a call.

```
1   begin int a;
2       proc p = (int i)void:
3                       a:=i;
4       begin int a;
5           a:=2;
6           p(a)
7       end
8   end
```

Figure 3.1

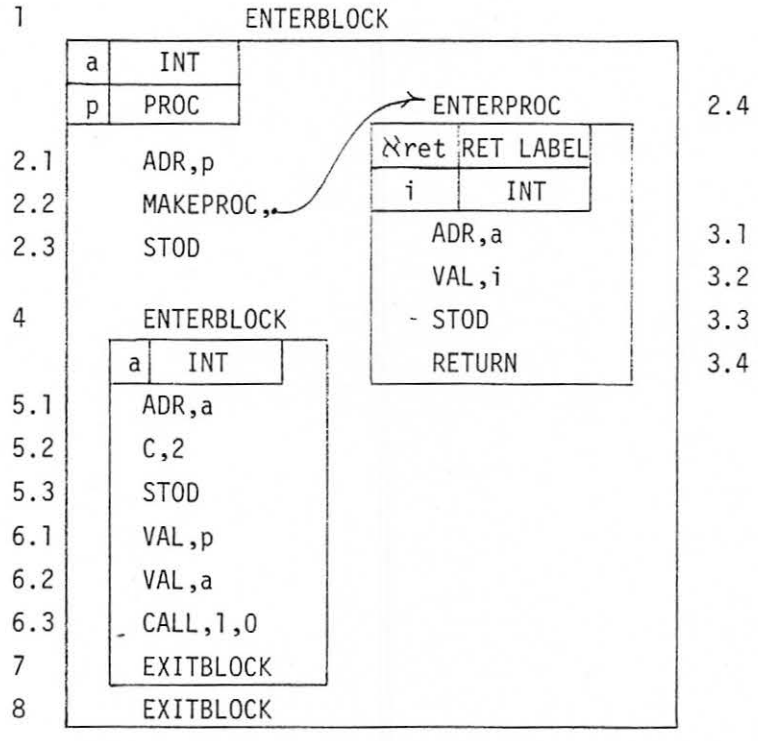


Figure 3.2

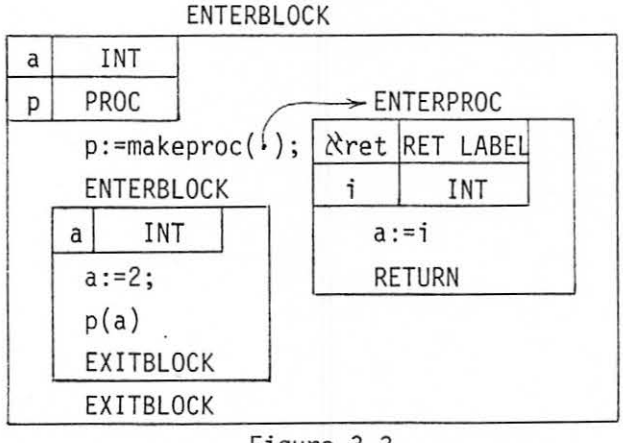


Figure 3.3

The instructions in most of our examples will be intermediate level polish style instructions. In this case each instruction corresponds to a basic semantic primitive, e.g., evaluation of a variable on the left hand side of an assignment, evaluation of the value of a variable or of a constant, performance of some operation such as the arithmetics, logicals, comparisons, assignment, etc., block entry or exit, procedure call or return, goto, etc.

In other cases, for the sake of compression, high level source language statements may be used as instructions. See figure 3.3 for a higher level rendition of the algorithm of figure 3.2.

The record of execution consists of a nest of record contours with a set of processors and processes. See the right half of figure 3.4 for the record of a snapshot in the computation of the algorithm of figure 3.2.

Each record contour is an activation of an algorithm contour, i.e., that of a block or a procedure. The algorithm contour is said to be the antecedent of the record contour, but since there may be more than one activation of a given algorithm contour (due to perhaps recursion or multiprocessing), the record contour is a descendent of the algorithm contour.

Each record contour, C, contains

1) a static link pointing to the record contour D nested about C; D is a descendent of the algorithm contour nested about C's antecedent. We typically do not show this link, as it is represented quite adequately by the graphical nesting of the contours
2) a value array consisting of subcells pairing the identifiers declared in C's antecedent with their values.

We depart from the usual CM and distinguish between processors and processes. The designations are level-relative; at a given level a processor is a self running processing unit capable of executing instructions and effecting the required changes to the memory; a process is a data cell not capable of executing but serves instead as a receptacle capable of remembering the state of a processor at some instant in time. It must be emphasized that this distinction is only level-relative, for a process at one level may be considered a processor at the next higher level.

If a model represents a "real" machine, then the processors will be the processing units of the machine, and their number will be fixed through all computations. In any case, the number of processes at any level can and does vary through a computation as they are allocated and deallocated. In all of our examples there will be precisely one processor, modeling the one CPU of the machine.

A processor consists of at least a site of activity and a stack. Other components may be introduced later. The site of activity is composed of

1) an instruction pointer, ip, pointing to the next instruction, i, to be executed by the processor, and
2) an environment pointer, ep, pointing to a record contour which is a descendant of the algorithm contour nested about the instruction i.

The stack is used to store the temporaries resulting from a polish evaluation of expressions and is composed of

1) a stack pointer, sp, pointing to a sufficiently large vector of subcells for storing the temporary values
2) a top_of_stack pointer, ts, pointing to the first free subcell on top of the stack.

We shall generally ignore the exact details of the implementation of the stack and instead shall consider only its abstract behavior as exhibited by the usual

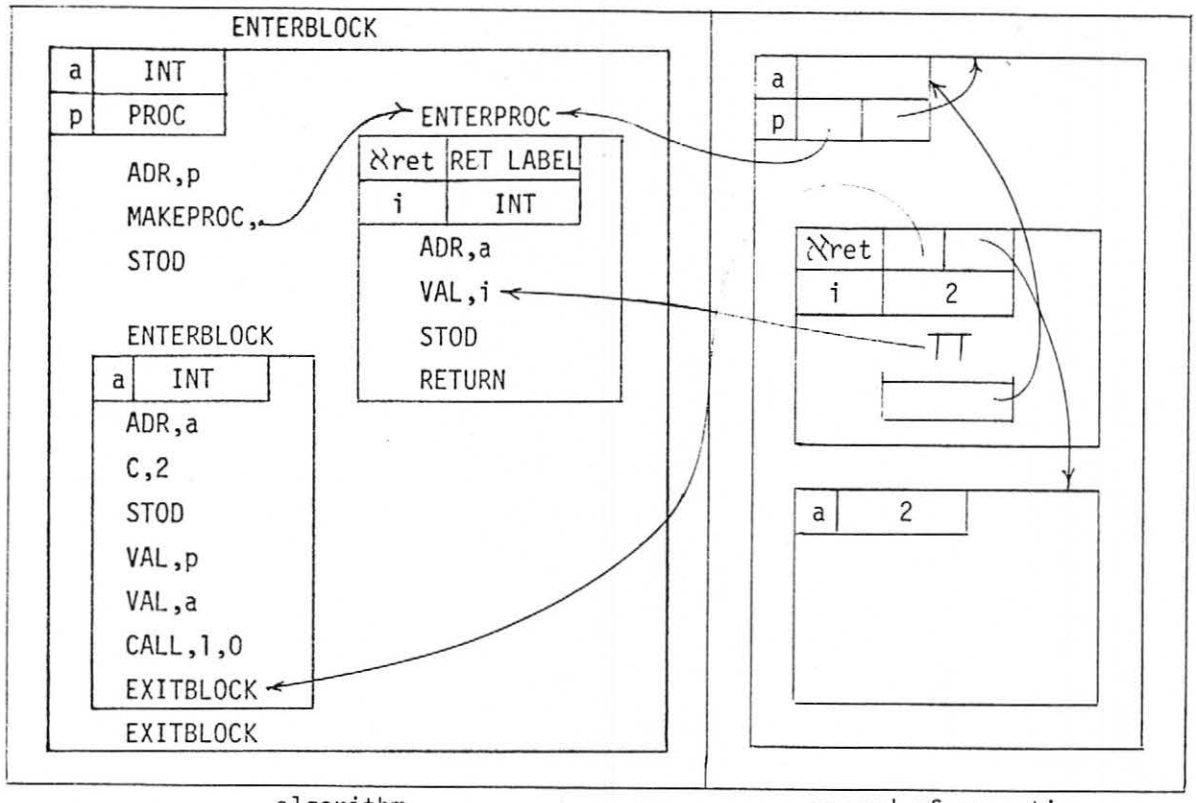


Figure 3.4

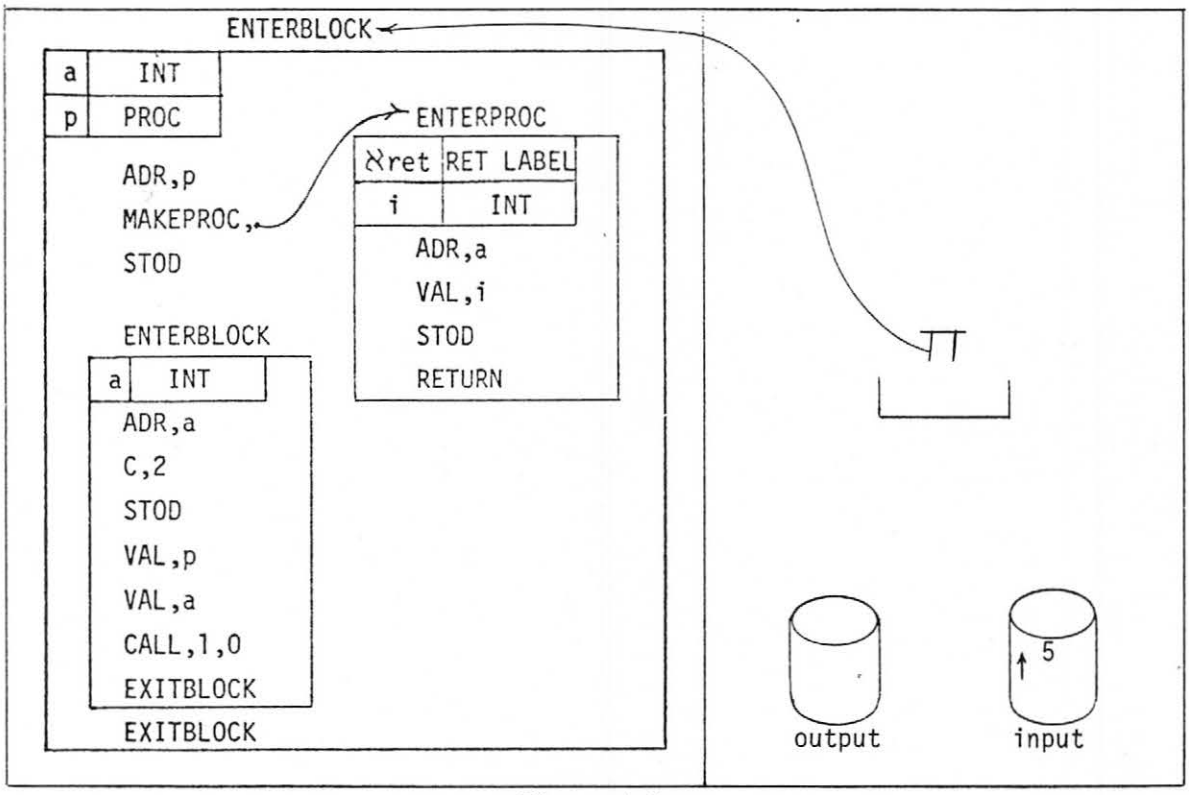


Figure 3.5

stack operations, push, pop, top, and is_empty [LZ74].

A process is a data cell consisting generally of the same components as a processor.

We shall typically denote a processor by a solid π and a process by a hollow In the interest of reducing pointer clutter (otherwise known as spaghetti), the ep of a processor or process will rarely be shown; instead the processor or process will be placed directly inside the contour to which its ep points. Also we shall eschew the sp of a processor or process by drawing the stack vector to which it points as a cup directly below the processor or process; additionally the ts shall be only implied by showing the vector up to and including the top subcell.

A processor or process π has an accessing environment. If π's ep is NIL the accessing environment is said to be empty. Otherwise, π's accessing environment is the list, $C_1,\ldots, C_n$, with $n\geq 1$ of record contours such that π's ep points to C_1; for $2\leq i\leq n$, C_{i-1}'s static link points to C_i; and C_n's static link is NIL. In other words, π's environment is the list of contours from inside-out that surround it. Anytime, in the course of instruction execution, a processor accesses an identifier, it uses the first occurrence of a subcell for the identifier in its accessing environment.

A cell (i.e., a record contour or a process) in the record of execution is said to be accessible to a processor if it is pointed to by a value in a processor or in a cell which is accessible to a processor. An allocated record cell remains allocated until it is no longer accessible to a processor, whereupon it may be deallocated.

The various I/O files used by the program in the algorithm are considered part of the record because they are time varying. We will assume all two of our files to be stream files [JW 74]. A file consists of a stream of characters including blanks of some finite length plus a cursor pointing to the next position from which to read or into which to write. The two files are the input and the output files. If a program is not doing input or output or both, we may not show the input or output or both files in the snapshot.

3.2 Values

Beside the usual set of boolean, integer, real, character, and pointer values, there are the procedure and label values.

A procedure value consists of all information, save for actual parameters, needed to do a call: the value has two components

1) an ip pointing to the entry point of the procedure body,
2) an ep pointing to a descendant of the algorithm contour of the block or procedure nested about the procedure body. This ep identifies the accessing environment containing the subcells for all nonlocal identifiers of the procedure body.

A label value consists of all information needed to move the processor to a new site of activity and to continue evaluation of whatever expressions the new site is in the midst of. Thus a label value is composed of

1) an ip pointing to the labeled instruction,
2) an ep pointing to a descendant of the algorithm contour of the block procedure nested about the labeled instruction,
3) a stack containing the temporaries needed to continue execution at the labeled instruction.

The ep of a procedure value and the ep and the stack of a label value are copies of those of the processor just after entry to the block containing the procedure

or the labeled statement.

3.3 Initial Snapshots and $init_{CM}$

Given a program p in the contour model algorithm language and an input list δ, $init_{CM}$ constructs the following initial snapshot

1) The algorithm is p.
2) The record contains only
   a) a single processor whose
      1) ip points to the first instruction of p
      2) ep is NIL
      3) stack is empty
   b) an input file whose stream is set to δ and whose cursor is set to point to the first position
   c) an output file whose stream is blank and whose cursor is set to point to the first position.

Figure 3.5 shows the initial snapshot formed from the program of figure 3.2 and the input stream " 5 ".

3.4 Transformation

The transformation for the single processor contour model is the following four-step procedure:

Let π be the processor.

1) If π's ip does not point to an instruction, i.e., it is NIL, then halt the computation.
2) Fetch the instruction inst pointed to by π's ip.
3) Sequence π's ip to point to the next instruction if there is one; if not set π's ip to NIL.
4) Execute inst.

The execute portion of the transformation is what causes the changes to the snapshow resulting from the performance of an instruction. We shall describe the execution of most instructions as they are introduced, but a few key instructions merit a brief description now.

ENTERBLOCK: Allocate a record contour for the algorithm contour being entered by the instruction, set the new contour's static link to a copy of π's ep, and reset π's ep to point to the new contour.

EXITBLOCK: Reset π's ep to a copy of the static link of the record contour that π's ep points to (this may leave a record contour inaccessible).

CALL, n, m: Assumes that the top values on the stack are actual parameters ordered from last to first, the n+1th value on the stack is a procedure value, and the procedure requires m local variables. Allocate a record contour with space for one return label, n parameters, and m local variables. Pop and pass each parameter value into the appropriate subcell of the new contour. Set the new contour's static link to a copy of the procedure's ep. Save π's current site of activity into the return label cell of the new contour (the ip already points to the instruction after the CALL). Reset π's ip and ep to that of the procedure value while replacing the procedure value in the stack by a reference to the new contour.

ENTERPROC: Reset π's ep to a copy of the reference to a record contour on top of the stack. Pop the reference.

RETURN: Reset π's ip and ep to those of the return label in the first subcell of the record contour pointed to by π's ep.

See [Joh71, Org73, CDMPS73, and OFP78] for examples of computations in the model.

4. The Implemented and Implementing Machine

In all the methods of supporting a machine, there is an implemented machine which is supported in some manner by an implementing machine. In this section we establish some notation that will be used to discuss all the methods. Recall that we are restricting ourselves in this paper to deterministic models.

4.1 Definitions

A. Implemented Machine

1) $Md = (Id, Id_0, Fd)$ is the implemented machine whose machine language is Ld.
2) The program executed by a computation in Md will usually be called Pd.
3) The Computation $Md(init_{Md}(Pd,\delta))$, for some input δ, consists of the snapshots $Sd_0, Sd_1, \ldots, Sd_i, \ldots$.

Sometimes the usage is sloppy and it names the implemented machine simply by its language. However there are, in general, many machine architectures capable of executing a given language, and while it is somewhat irrelevant to the user of the language what the architecture is, it makes all the difference in the world to the implementor of the language what architecture is to be implemented.

To allow easier comparison of the methods of support, in all of our examples, Md will be the CM with a postfix polish machine language Ld. The meaning of each instruction of Ld either will be obvious or will be explained as it is introduced.

B. Implementing Machine

1) $Mg = (Ig, Ig_0, Fg)$ is the implementing machine whose machine language is Lg.
2) The program executed by a computation in Mg will usually be called Pg.
3) The computation $Mg(init_{Mg}(Pg,\delta))$ for some input δ consists of the snapshots $Sg_0, Sg_1, \ldots, Sg_i, \ldots$.

The implementing machine in all our examples will be the CM executing either

1) a high level Algol 68-like language
or 2) a post-fix polish language.

The preference will be to use the former unless the discussion of the method requires the post-fix polish language. Any construct used in either kind of language either will be self-explanatory or will be explained as it is used.

4.2 Formal Requirements for Methods of Supporting Machines

All methods of supporting Md by Mg may be described by a single commuting diagram

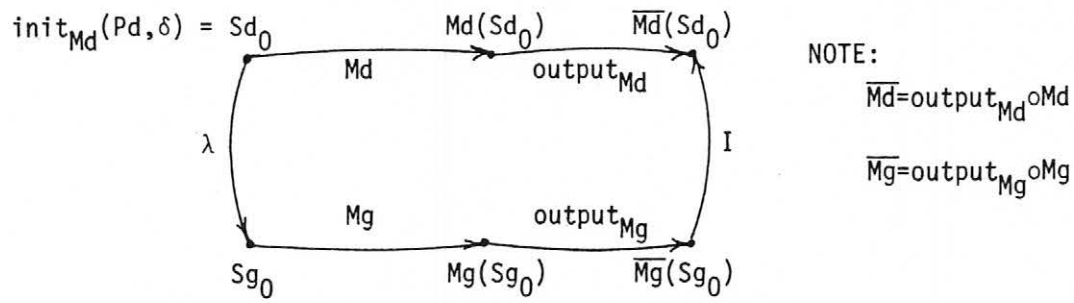


Figure 4.1

We are given the implemented machine Md. A method of supporting Md may be characterized by a loading mapping λ, the implementing machine Mg, and an equivalence criterion.

The loading mapping

$$\lambda: Id_0 \rightarrow Ig_0$$

maps the implemented initial snapshots to the implementing initial snapshots and embodies

1) compilation, if any, of the source program Pd and
2) setting up some representation of the implemented initial snapshot in the various parts of the implementing initial snapshot.

We shall take as the equivalence criterion the identity mapping I on the output of the implementing computation to the output of the implemented computation.

Thus a method of support for Md is simply a pair (λ, Mg).

It is required for any method (λ, Mg) for supporting Md that

for all Pd in Ld and
for all δ in INPUT,

$$\overline{Md}(init_{Md}(Pd,\delta)) = \overline{Mg}(\lambda(init_{Md}(Pd,\delta)))$$

i.e. that

$$\overline{Md} = \overline{Mg} \circ \lambda$$

Stated simply, it is required that the implementing computation produce the same result as the implemented computation, i.e. either both do not halt or both halt and produce the same output.

However, in practice, it turns out that this simple requirement is too weak.

1) Proving that the equation holds is difficult because it requires that a whole computation be dealt with as if it were one step.
2) It is only by looking at the outputs of intermediate steps of the computations that two nonterminating computations can ever be judged as different
3) More insight into the difference between the methods can be gained by comparing corresponding snapshots in the implemented and implementing computation.

Therefore we prefer to use a stronger inductive statement of the requirements which implies the holding of the weaker requirements given above.

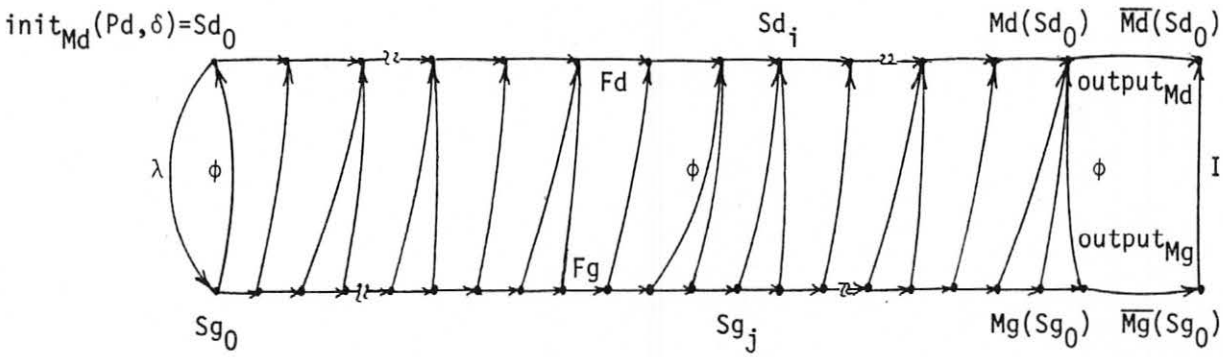


Figure 4.2

We require for each method (λ,Mg) of supporting Md that there exists a mapping ϕ: Ig → Id such that for all Pd in Ld and all δ in INPUT, if $Sd_0 = init_{Md}(Pd,\delta)$,

1) $\phi(\lambda(Sd_0)) = Sd_0$
2) <u>If</u> $\phi(Sg_i) = Sd_j$ <u>and</u> Sd_j is not a final snapshot

then $\exists$ finite $m \geq 1$ such that
a) $\phi(Sg_{i+m}) = Sd_{j+1}$
and b) $output(Sg_{i+m}) = Sd_{j+1}$

3) If $\phi(Sg_i) = Sd_j$ and Sd_j is a final snapshot
then $\exists$ finite $m \geq 0$ such that
a) Sg_{i+m} is a final snapshot
b) $\phi(Sg_{i+m}) = Sd_j$
and c) $output(Sg_{i+m}) = output(Sd_j)$.

That is, we require that there is a mapping ϕ which allows the implemented snapshots to be extracted from the implementing snapshots in such a way that

1) The initial implemented snapshot may be obtained from the initial implementing snapshot, and the outputs of the two snapshots are the same.
2) If at some point an implementing snapshot Sg_i maps to a nonfinal implemented snapshot Sd_j, then at most some finite number of implementing computation steps will yield a snapshot Sg_{i+m} which maps to the next implemented snapshot Sd_{j+1} and which has the same output as Sd_{j+1}.
3) If at some point an implementing snapshot Sg_i maps to a final implemented snapshot Sd_j (which, of course, is the final snapshot of its computation), then at most some finite number of implementing computation steps, including none, will yield the final snapshot Sg_{i+m} of the implementing computation which maps to Sd_j, the final implemented snapshot, and which has the same output as Sd_j.

This formulation of the requirements is clearly stronger than is required just to be able to say that the implementing computation produces the same result as the implemented computation. It says that the implementing computation simulates the implemented computation step-by-step. The latter implies the former but not vice versa.

Note that this formulation of simulation assumes that the implementing computation requires at least one step to simulate one step of the implemented computation. This assumption causes no difficulty, for, as we shall see, all of our implementing models do require at least one step and possibly more to do one step in the implemented model.

If all we are interested in is that the implementing model produce the same result as the implemented model then the pair (λ, Mg) suffices to characterize a method of support. If we insist that the implementing model in some sense simulates the implemented model then λ, Mg, and ϕ are needed to characterize a method. Because examination of ϕ does shed some light on the nature of the methods of support and they are all in some sense simulations, we use the latter means of characterizing a method of support.

Accordingly, in the remaining sections, as a method is described we shall describe informally Mg, the implementing model, λ, the loading map, and ϕ, the implemented snapshot extraction map.

5. Interpretation

In interpretation, the snapshot of the implemented machine is contained entirely as a data structure within the record of execution of the snapshots of the implementing machine. In particular, the processors of the implemented machine become

processes in the snapshot data structure. The processor of the implementing machine, under direction of a program called an interpreter, manipulates its representation of the snapshot of the implemented machine to reflect the computation in the implemented machine.

5.1 Single Process Interpretation

In the case that the implemented machine has a single processor, the interpreter program in the algorithm of the implementing machine is of the following form (In the following

X*

means "X dereferenced" and

X.C

means the "C component of X"):

```
begin
    ref process current;
    instruction inst;
    ref snapshot snap;
     .
     .
     .
    while current*.ip≠nil do
        fetch    :  inst ← current*.ip*;
        increment:  current*.ip ← current*.ip+length(inst);
        execute  :  case inst.opcode in
                     .
                     .
                     .
                     'ADR'    : ¢ pushes address of variable whose identifier
                                  is arg1 of inst ¢
                                push (current*.stack, reference to cell for
                                (inst.arg1)),
                     'READINT': ¢ assumes reference to stream input file is on
                                  top of stack, replaces reference by next integer
                                  in stream ¢
                                infile ← top(current*.stack);
                                replace top(current*.stack, the next integer from
                                  (infile)),
                     'STOD'   : ¢ assumes value on top of stack and reference
                                  below that, assigns value to referred to cell,
                                  pops value and reference  ¢
                                2nd (current*.stack)* ← top(current*.stack);
                                pop twice (current*.stack),
                     'VAL'    : ¢ pushes value of variable whose identifier is
                                  arg1 of inst ¢
                                push (current*.stack, value of cell for
                                    (inst.arg1));
                     'HALT'   : ¢ set ip to nil ¢
                                current*.ip ← nil
                     .
                     .
                     .
                     esac
    od
     .
     .
     .
end
```

Figure 5.1

The interpreter declares at least the following variables (which in the case of an emulation would be assigned to registers of the host machine):

1) current to contain a reference to the process representing the implemented processor
2) inst to contain the currently executed instruction
3) snap to contain a reference to the area containing the entire snapshot of the implemented machine. This area contains the process referred to by current

The main part of the interpreter is a loop which repeatedly has the processor take the process through a fetch, increment and execute cycle. Once the next instruction has been fetched and the process's ip advanced, a case determines which instruction to perform and directs the processor in modifying the snapshot referred to by snap in accordance with the semantics of the selected instruction.

Figures 5.3 through 5.5 show a nonconsecutive sequence of skeletal snapshots from the implementing computation supporting the implemented computation of the program fragment shown in figure 5.2.

```
                ENTERBLOCK
                DECLARE,n,INT      ¢ assembly pseudo instruction ¢
                DECLARE,sysin, REF FILE
                   .
                   .
                   .
ALGORITHM       ADR, n             ¢ get address of n ¢
CONTOUR C1      VAL, sysin         ¢ get reference to file sysin ¢
                READINT            ¢ replace ref with next integer in sysin
                                     stream ¢
                STOD               ¢ assign integer to n ¢
                   .
                   .
                   .
                EXITBLOCK
```

Figure 5.2

In this case, λ simulates starting up the interpreter, loading the program to be interpreted, setting the (interpreted) process's ip to point to the first instruction of the program, and leaving the (interpreting) processor ready to begin a fetch, increment, and execute cycle.

1) In the algorithm is the interpreter.
2) In the record are
   a) the processor sitting inside a record contour for the outer block of the interpreter ready to execute the statement labeled fetch
   b) a representation of the initial implemented snapshot with
      1) the processor of the implemented snapshot turned into a process with identical content, and
      2) the files of the implemented snapshots represented by like initialized files of the implementing snapshot.
   c) In the record contour for the outer block of the interpreter,
      1) current is initialized with a pointer to the process in the representation of the implemented snapshot
      2) snap is initialized to point to the representation of the implemented snapshot.

See figure 5.6 for a schematic diagram of this loading map.

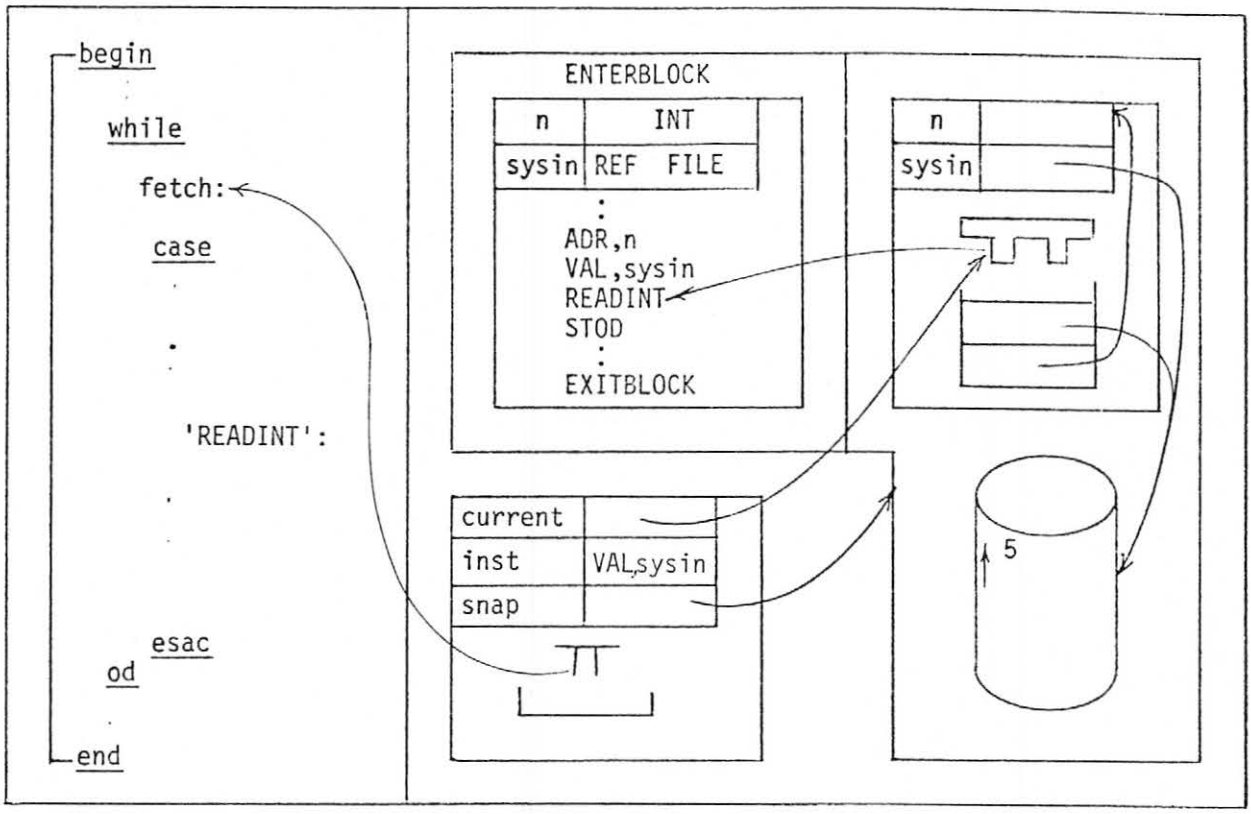


Figure 5.3

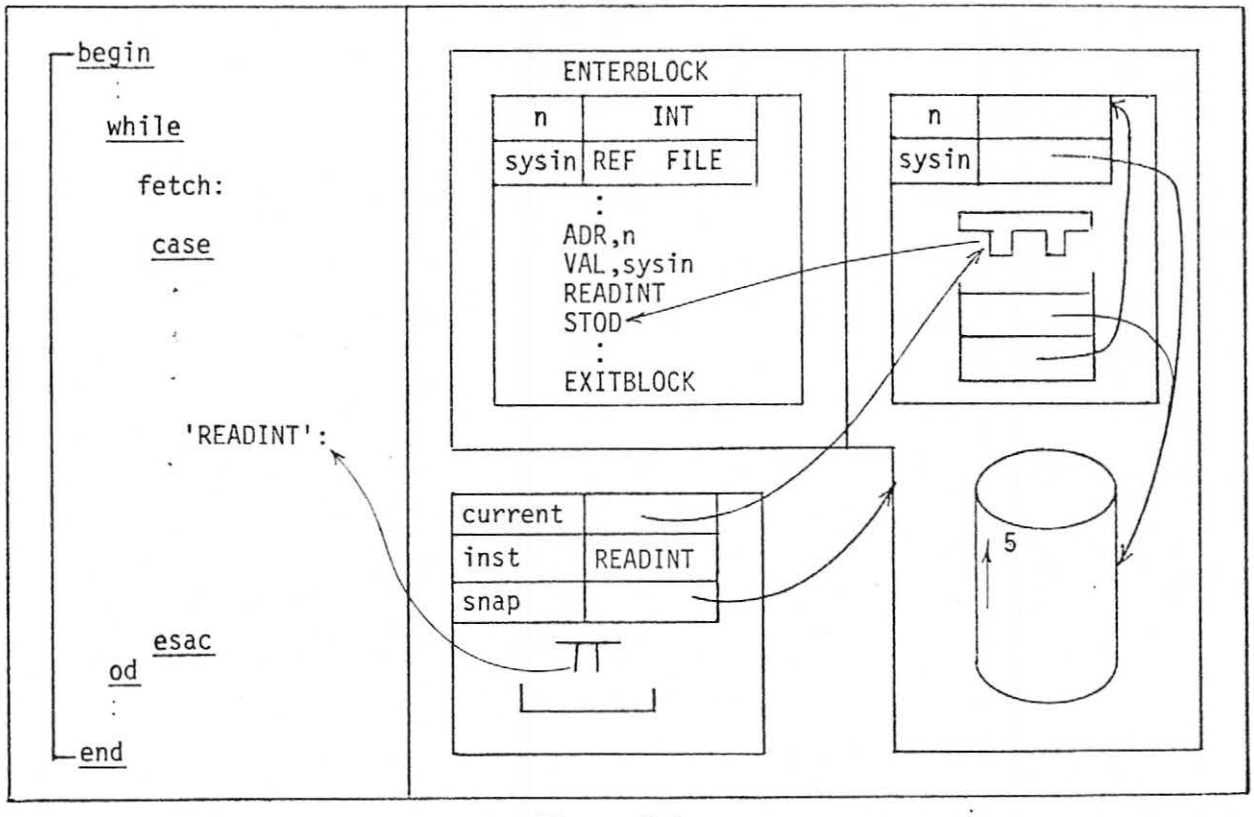


Figure 5.4

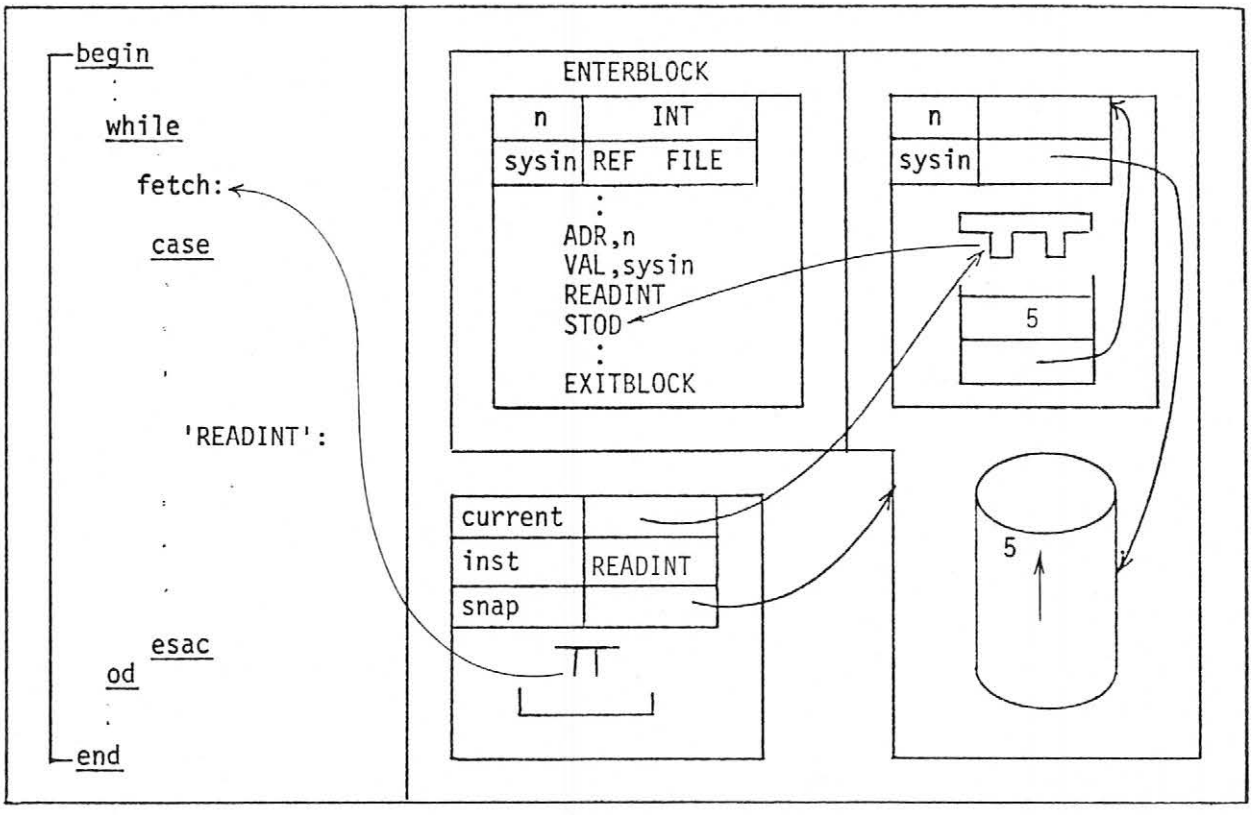


Figure 5.5

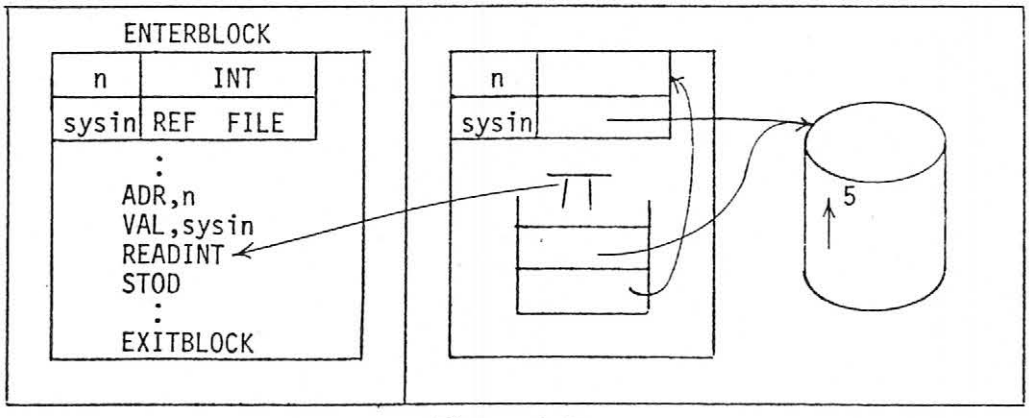


Figure 5.7

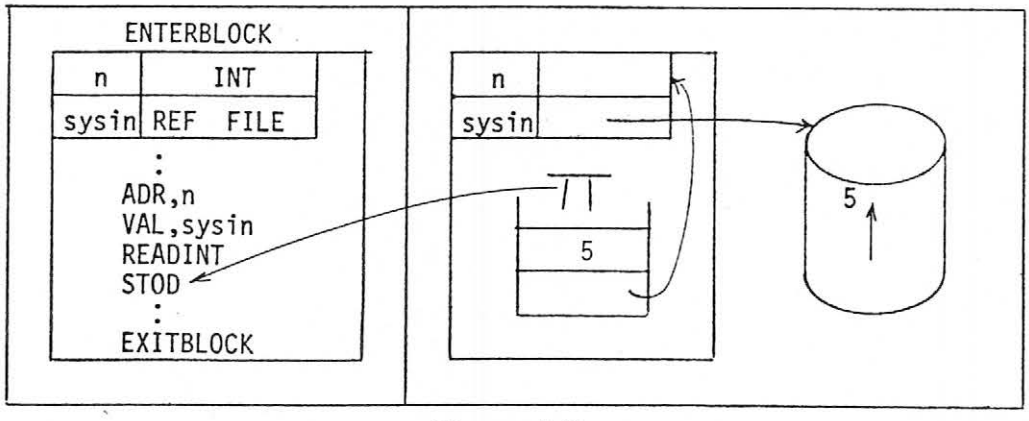


Figure 5.8

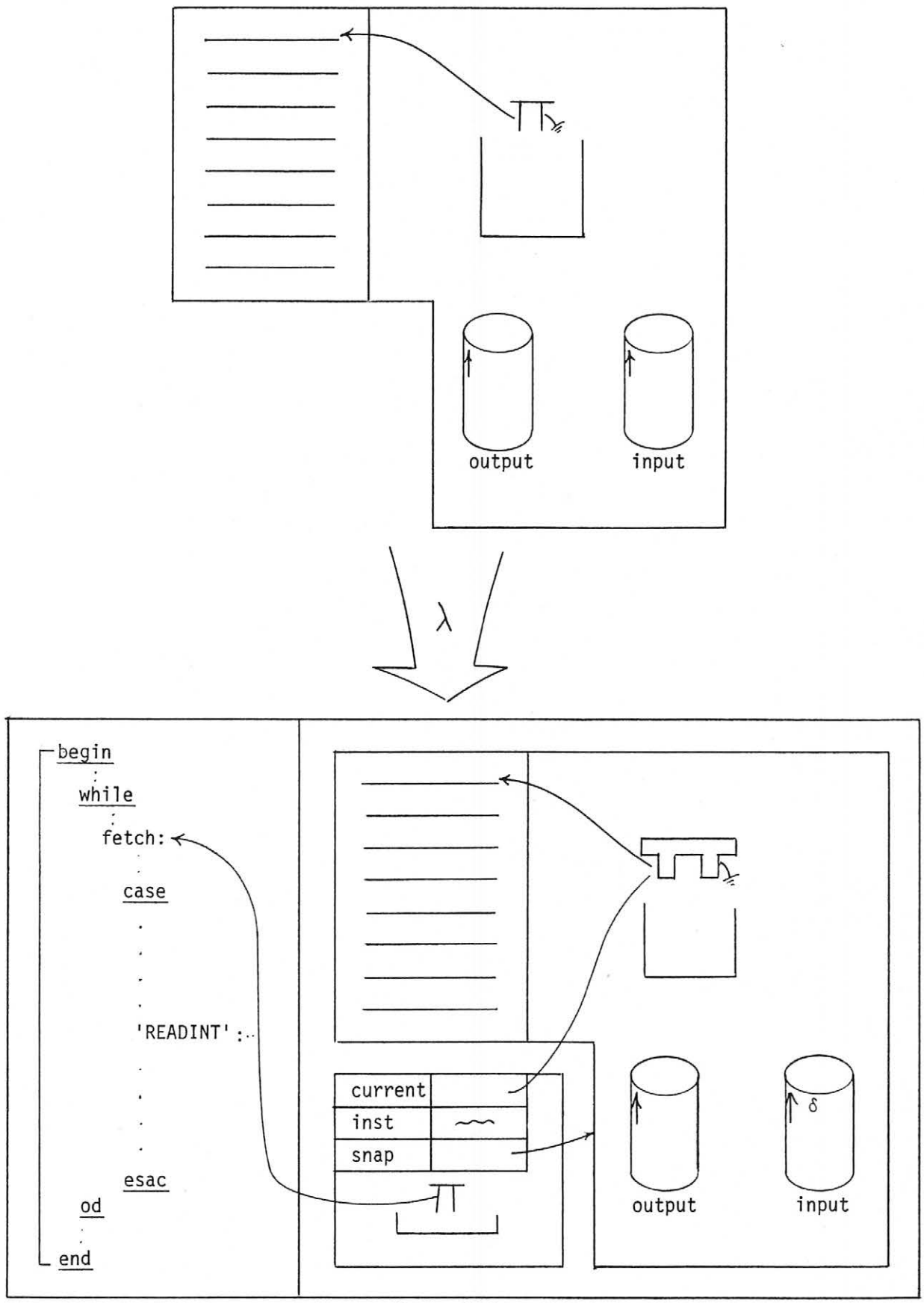

output
input
λ
begin
while
fetch:
case
'READINT' :
esac
od
end
current
inst
snap
output
input
δ

Figure 5.6

The ϕ mapping may be viewed* as follows: Consider the implementing computation

$$C_g = \langle Sg_0, Sg_1, \ldots, Sg_i, \ldots \rangle.$$

Take in order of appearance in Cg those snapshots in which the processor's ip points to the statement labeled fetch to obtain the sequence

$$C_g' = \langle Sg_0', Sg_1' \ldots, Sg_j', \ldots \rangle$$

Then form the implemented computation

$$Cd = \langle Sd_0, Sd_1, \ldots, Sd_j, \ldots \rangle$$

of the same length as C_g' such that for all j, $0 \leq j \leq \text{length}(C_g')$,† Sd_j is that part of Sg_j pointed to by the value of the snap variable with the process turned into a processor of identical content.

Consequently, applying this ϕ to the sequence of snapshots embodied by figures 5.3 to 5.5, all snapshots but those of figures 5.3 and 5.5 would be dropped to form C_g', and these would be converted by ϕ to the implemented skeletal snapshots shown in figures 5.7 and 5.8.

6. Enmasterization

In enmasterization, the implemented language is called the user language, and the implementing language is called the master or supervisor language. In the typical computing system, it is not desirable to give to the user and his or her language direct control over some or all the resources of the system, e.g., the processor, the various input/output devices, etc. Consequently, on machines supporting such systems, instructions giving the user direct access to these resources e.g., load processor, start read, start write, etc., are made privileged. A user program is not allowed to execute privileged instructions, but a master or supervisor program is allowed to execute privileged instructions. In order for the user program to access one of these resources, it must somehow request the supervisor to access the resource on its behalf, performing the resource related operation, and reporting back to the user program when the operation is completed.

One method of requesting supervisor help requires one processor and no processes.

6.1 No-process enmasterization

In no-process enmasterization, the user program executes a special supervisor call instruction whose argument tells which operation the supervisor is to perform on its behalf. As a result of the supervisor call, a trap occurs bringing control to the supervisor. A trap is roughly a procedure call combined with a processor mode change - from user to master mode. Control returns to the user program at the instruction after the supervisor call by means of a special return which also changes the processor's mode - from master to user mode.

Also causing traps are such events as time slice end, illegal op code, attempted use of privileged instruction by user, etc.

To model this phenomenon, in the algorithm of the implementing machine must be a

*Strictly speaking ϕ should not be defined on a computation to a computation, but rather on a snapshot to a snapshot. However, it is easy to construct the correct ϕ from the one described. If π.ip in the implementing snapshot does not point to the statement labeled fetch, the true ϕ includes the finite process of executing in Mg until π.ip does point to the statement labeled fetch. We will use this slightly incorrect but convenient view of ϕ from now on.

†If C_g' is not finite then length (C_g') is taken as ∞.

special supervisor routine. This routine examines the argument of the supervisor call and branches to the supervisor subroutine which performs the requested operation using whatever instructions, privileged or otherwise, are necessary to do the job. (For simplicity, we assume that the argument of the supervisor call is the ip of the selected subroutine; if desired, a simple branch table can easily be added to make the argument values independent of where the subroutines happen to be.) At the end of each of these subroutines is a return to the supervisor caller.

There is exactly one processor, and beside the basic site of activity and stack components, the processor has a U/M bit indicating whether the processor is in user or master mode. The processor may execute a privileged instruction only in master mode.

A place is needed to store a procedure value denoting the supervisor routine. The processor must know where this place is so it can do the supervisor call, but it is desirable, for reasons of security, that this place be invisible to the user. We store this procedure in a special register of the processor, the supervisor procedure value register (our diagrams show this at the bottom of the stack).

Both the user routine and the supervisor routine may need to call some system utility routines which do not use privileged instructions. These must be declared in a place visible to both the user and the supervisor routines. We therefore assume that there is a system block declaring and initializing these routines. The user program's outer block and the supervisor routine are nested inside this block. Over in the record, all user program record contours and supervisor routine record contours are nested inside a record contour for the system block. In the subcells of this contour are the (ip,ep) pairs for the various system utility routines.

If we assume that an ordinary procedure call and return do not change the mode of a processor, then we have the possibility of keeping the supervisor routine which contains all the instances of privileged instructions as small as possible. A section of code which forms a widely useable module and which does not use any privileged instruction may be made an ordinary procedure separate from the supervisor routine but callable from it and the user routines.

In figure 6.1, we show fragments of a user program and of the supervisor routine for a possible system.

```
    user code
      ENTERBLOCK
      DECLARE, n, INT
      EQU, readint, some integer
       .
       .
       .
      ADR, n
      SVC, readint          ¢ supervisor call ¢
      STOD
       .
       .
       .
      EXITBLOCK
    supervisor routine:
entrypoint:   ENTERPROC
              DECLARE, mode, BIT
              DECLARE, code, IP
              VAL, code  ¢ get SVC code ¢
              BRANCH     ¢ use code as address to branch to ¢
                   .
                   .
```

```
readint:    ALLOC,80          ¢ allocate buffer, leave pointer to buffer in stack ¢
            STARTREAD         ¢ when it's done, the input is in buffer pointed to
                                by reference on top of stack ¢
            WAITREAD          ¢ loop on this instruction until read is done ¢
            VAL, extract int  ¢ push procedure value for system procedure
                                extract int ¢
            ALLOC_CALL_CONTOUR  ¢ and call this procedure ¢
            CALL              ¢ which replaces reference to string on top of stack
                                by integer it finds in string ¢
            SRETURN           ¢ return from supervisor call ¢
              .
              .
              .
```

Figure 6.1

In these fragments

1) SVC,x is the supervisor call; its execution proceeds as follows: A contour with subcells for a return label, a mode, and a code is allocated and linked to the system environment which declares the various system procedures, e.g., extractint. The processor's current site of activity (its ip already points to the next instruction) and its current mode are saved in the return label and mode subcells of the contour. The instruction argument x is stored in the code subcell. The processor then sets its mode to M and does a goto to the entry point of the supervisor routine using the (ip,ep) pair in the supervisor procedure value register.
2) SRETURN, a privileged instruction, returns from a supervisor call: The processor simply resets its site of activity and mode from the return label and mode subcells of the contour pointed to by its current ep. As a result of this instruction, this contour may become inaccessible and thereby be deallocated.
3) STARTREAD and WAITREAD are privileged instructions which initiate a read and wait until the read is done. When the read is done, the record read is stored in the buffer referred to by the pointer on top of the stack.

Observe that because a fresh contour is allocated for each supervisor call, the processor's current mode is saved in this contour, and SRETURN restores the processor's mode from the saved mode, supervisor calls may occur even in the supervisor routine or in routines called from it; even recursive supervisor subroutines are possible.

As a consequence of the development above, it may be seen that the implemented machine, whose language Ld comprises only the nonprivileged instructions, including the supervisor call, is really a submachine of the implementing machine, whose language Lg comprises all the instructions, including the privileged ones. Thus

$$Ld \subset Lg$$

i.e. the user language is a proper subset of the master language. The set Lg-Ld includes all the possible supervisor routines.

From the point of view of the implemented machine, each supervisor call with a different argument must be considered a separate instruction whose effect in one computation step is the same as that of the selected supervisor subroutine in the implementing machine. Thus for example in the implementing machine implied by the user program of figure 6.1, SVC, readint is a single instruction which gets one integer from the next input record and pushes that integer into the processor's stack.

Figures 6.2 through 6.5 show a sequence of some skeletal snapshots from the

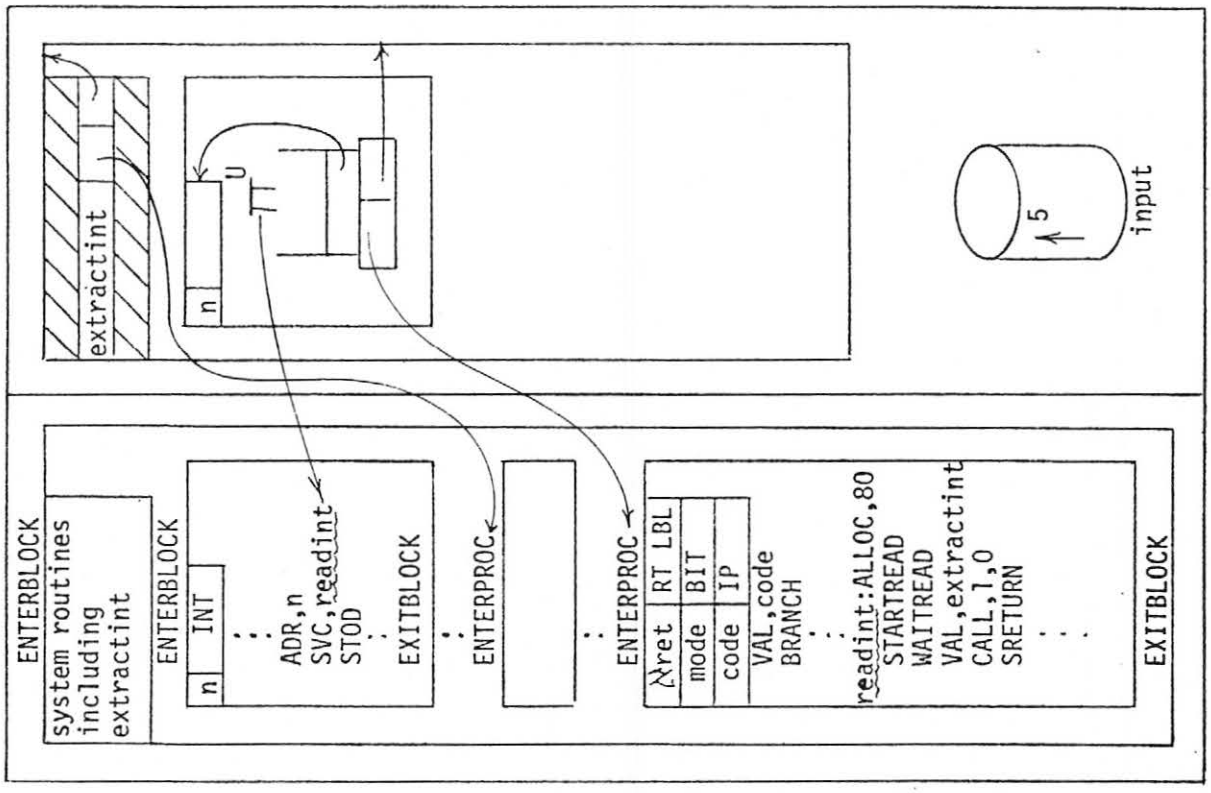


Figure 6.2

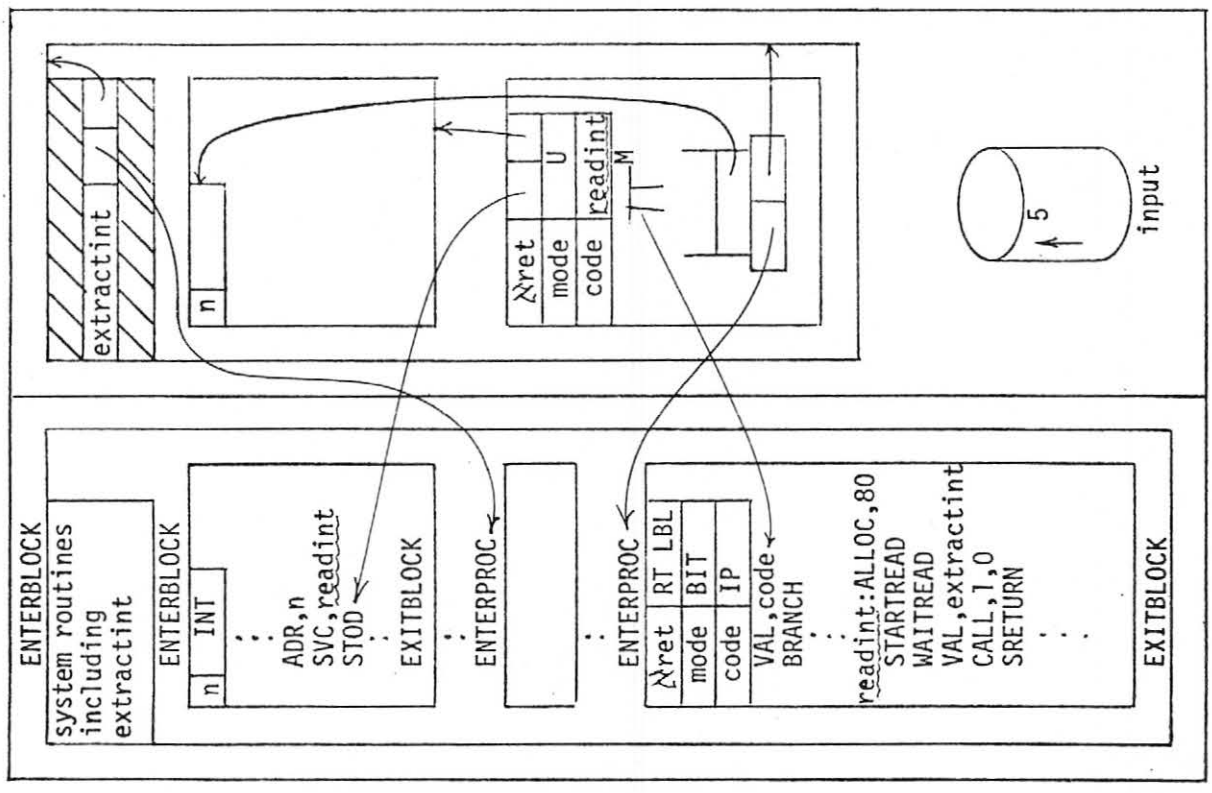


Figure 6.3

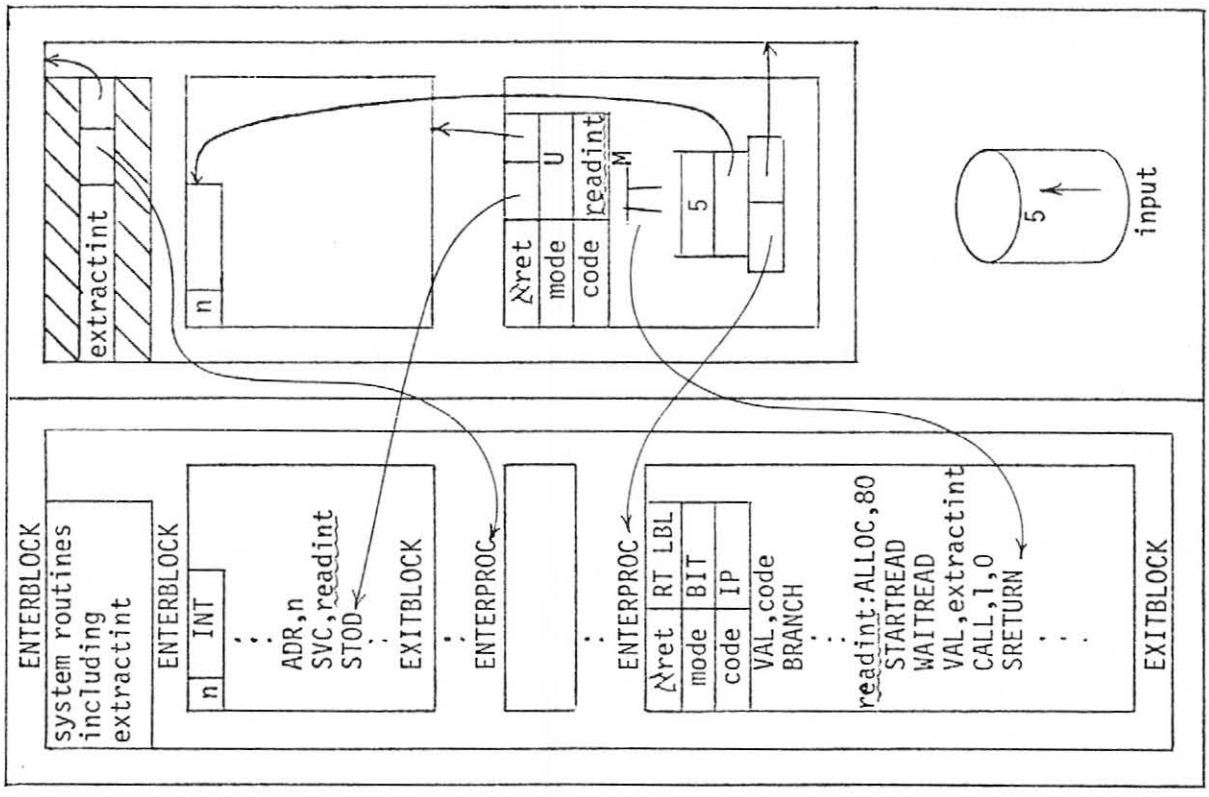


Figure 6.4

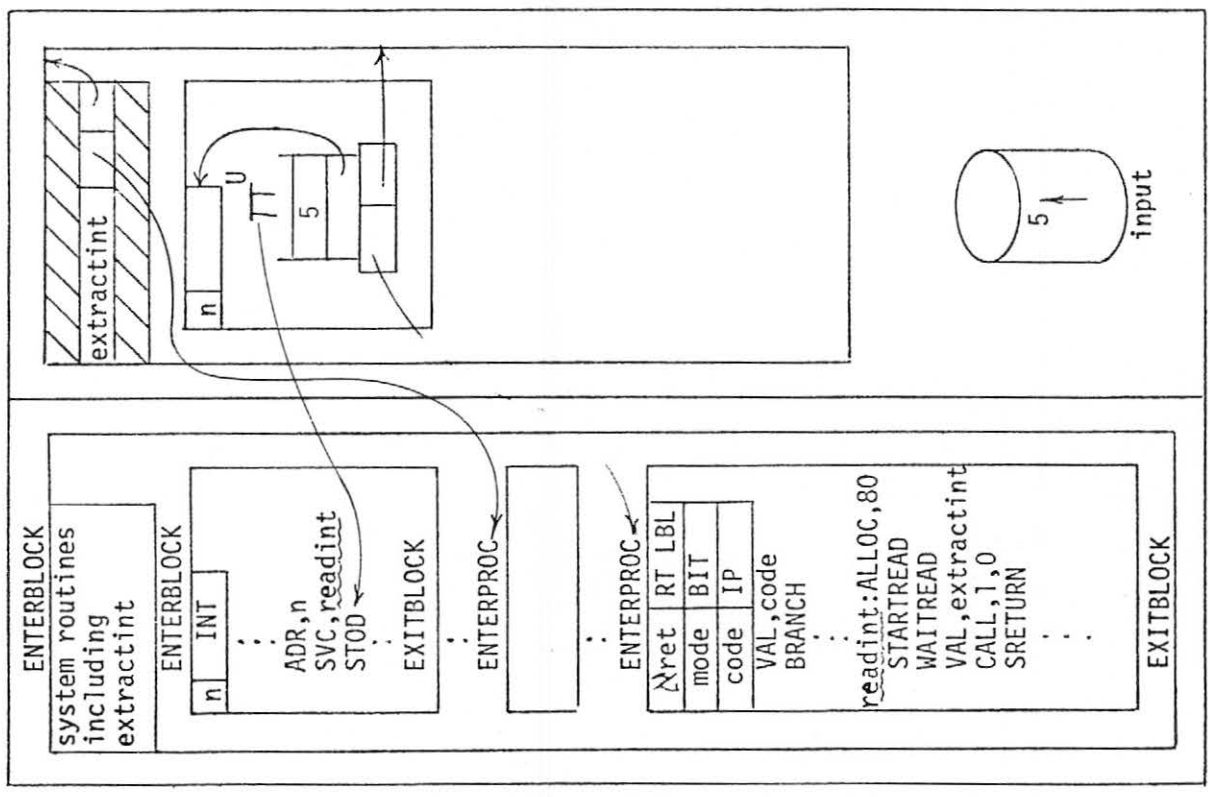


Figure 6.5

computation of the program fragments of figure 6.1.

It is useful to observe that the implementing machine is itself supported by a lower level meta-machine; it is in this meta-machine that the procedure call-like response to the supervisor call is programmed.

For the construction we have given for no-process enmasterization, the loading map λ is quite straightforward. However, we must first modify the init function for the implemented machine. Since the various nonprivileged system routines are written in the implemented language Ld and are callable from programs in Ld, the implemented machine must also have the system outer block. Examining the top half of figure 6.6, we see

1) in the algorithm, the system outer block declaring and initializing the system routines and containing the user program
2) in the record, the processor sitting inside a record contour for the system outer block ready to execute the first instruction of the user program. The subcells of the record contour are initialized with (ip,ep) pairs for the system routines. Also in the record are the usual input and output files.

This is the initial implemented snapshot.

The loading merely

1) Inserts the supervisor routine algorithm contour and code into the system outer block algorithm contour.
2) Adds a supervisor procedure value register to the processor and initializes it with an ip pointing to the supervisor routine entry point and an ep pointing to the system outer block record contour.
3) Sets the processor to the user mode.

See figure 6.6 for an illustration of this λ mapping.

The ϕ mapping for the two models is equally as straightforward. Consider the computation

$$Cg = Sg_0, Sg_1, \ldots, Sg_i, \ldots$$

in the implementing machine. Remove from this sequence all snapshots in which the processor is in master mode to obtain the sequence

$$Cg' = Sg'_0, Sg'_1, \ldots, Sg'_j, \ldots$$

Then form the implemented computation

$$Cd = Sd_0, Sd_1, \ldots, Sd_j, \ldots$$

from Cg' by taking each Sg'_j and

1) removing the supervisor procedure value register from the processor,
2) removing from the system outer block code the code for the supervisor routine,
3) removing the mode indication (which is necessarily U) from the processor.

to obtain Sd_j.

Applying the first part of this construction to the sequence of implementing snapshots embodied by figures 6.2 through 6.5 eliminates all but the snapshots in figures 6.2 and 6.5. Applying the rest of the construction to these yields the skeletal implemented snapshots of figures 6.7 and 6.8.

Note, finally that for enmasterization, ϕ is almost one-to-one reflecting the fact that most of the implemented machine's instructions are executed directly in the implementing machine. Contrast this with interpretation where ϕ is many-to-one reflecting the fact that many interpreter steps are needed to advance one step in

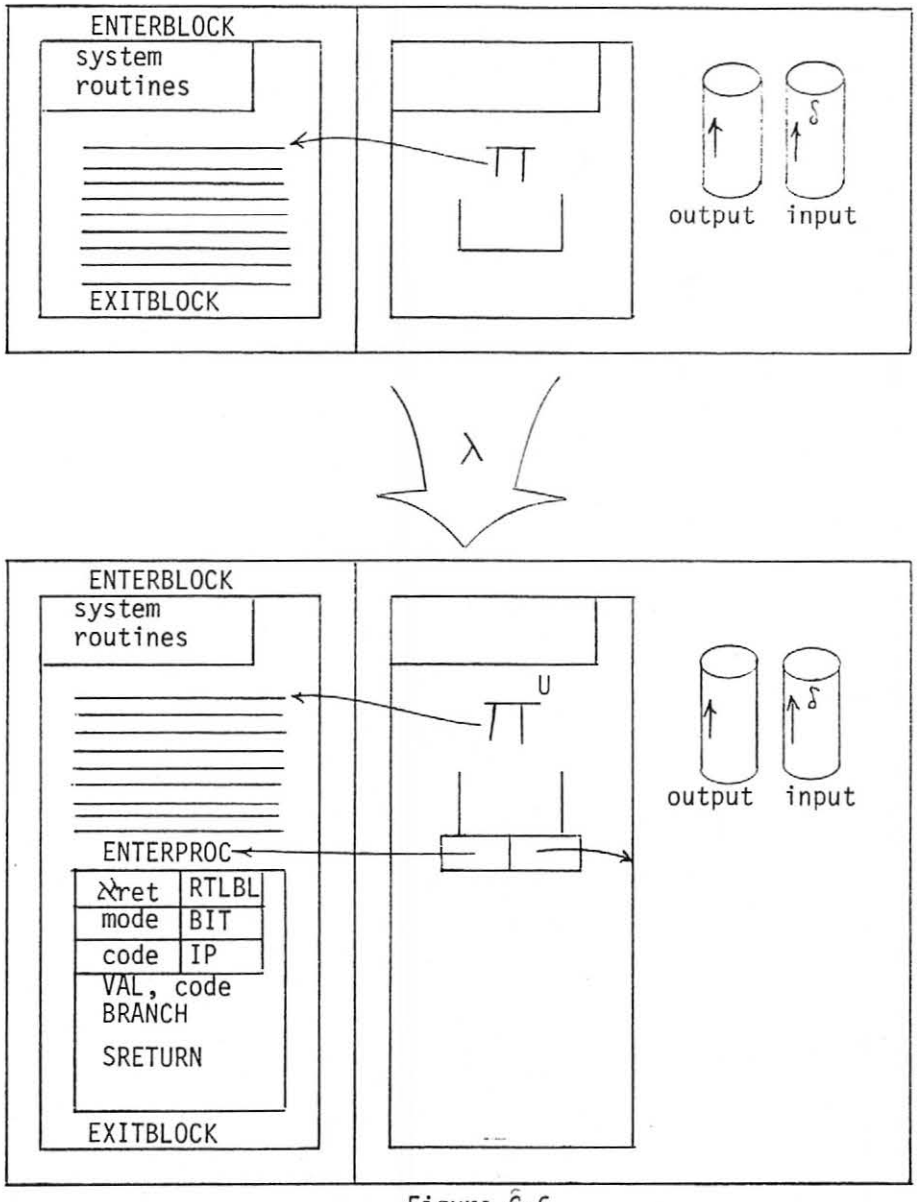

ENTERBLOCK
system routines
EXITBLOCK
output
input
λ
ENTERBLOCK
system routines
U
output
input
ENTERPROC
RTLBL
mode
BIT
code
IP
VAL, code
BRANCH
SRETURN
EXITBLOCK

Figure 6.6

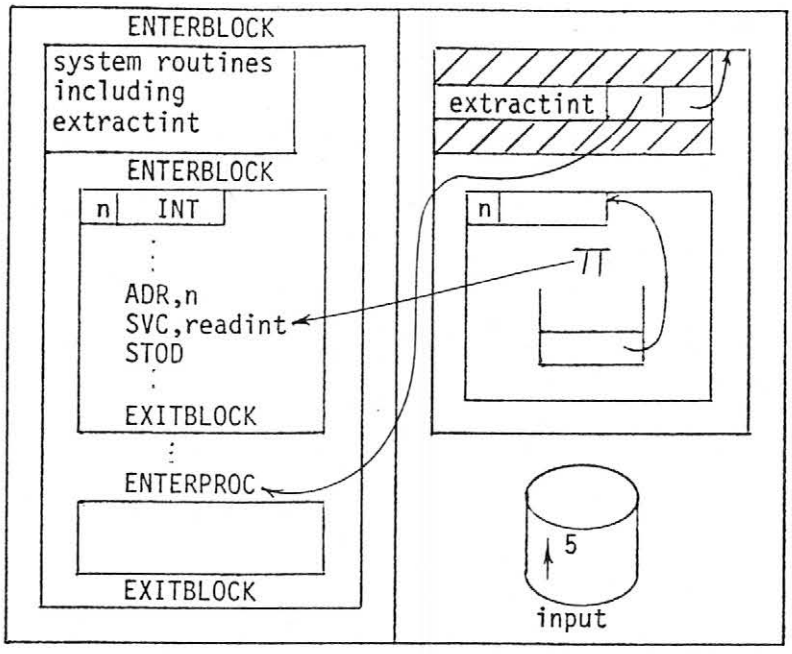


Figure 6.7

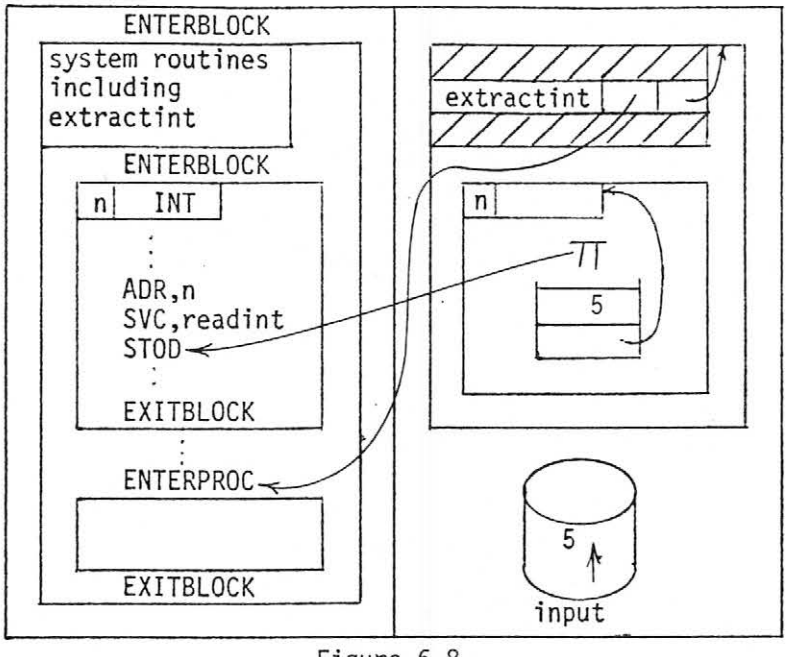


Figure 6.8

the implemented machine.

8. The Tower of //SYSBABEL [Bib??, Sam 69, MLB76]

In this final chapter, we reconsider the entire multilevel system given in figure 1.1 and give a new view of it.

First consider any two consecutive levels of the system. There is a ϕ map from the snapshots of the lower (implementing) level to the corresponding snapshots of the upper (implemented) level.

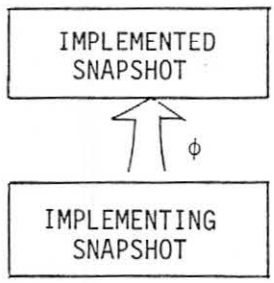


Figure 8.1

Now topologically contort the lower level snapshot so that the upper level snapshot may be physically superimposed on the contorted snapshot so that each implemented component lies on top of its implementing components. That is, in the superimposition:

1) given a component x in the lower level snapshot, x is at least partially covered by each component y of the upper level snapshot in whose construction under ϕ x participates.
2) given a component y in the upper level snapshot, y at least partially covers each component x of the lower level snapshot which participates in y's construction under ϕ.

This contortion may be a bit contrived and tortuous especially for support methods involving compilation and, in any case, if the lower level snapshot represents an intermediate state in the transition from an upper level snapshot to the next. However, in principle this contortion should always be possible.

For example, the application of the ϕ map for single process interpretation converts the snapshot of figure 5.3 to that of figure 5.7. This conversion results in the superimposition shown in figure 8.2.

In this kind of a superimposition any vertical line that cuts both snapshots passes through an implemented cell and its representation in the implementing snapshot.

In general, the lower level snapshot will be physically larger than the upper, as the lower level snapshot usually has extra code and data, e.g., an interpreter and its own variables, which help the implementation but which do not directly represent anything in the upper level. In no case, will the lower level snapshot be smaller than the upper.

Carrying this superimposition to the entire multilevel system of figure 1.1, we get something similar to our Tower of //SYSBABEL shown in figure 8.3. In it any vertical line cutting through all of the snapshots cuts through a cell in the LISP interpreter snapshot at the top level and such succeeding lower level's representation of it. The lowest level computation is moving the fastest; each lower level must do, perhaps, many steps to push the next higher level through one step; and the highest level is moving the slowest. The picture is that of a multi-

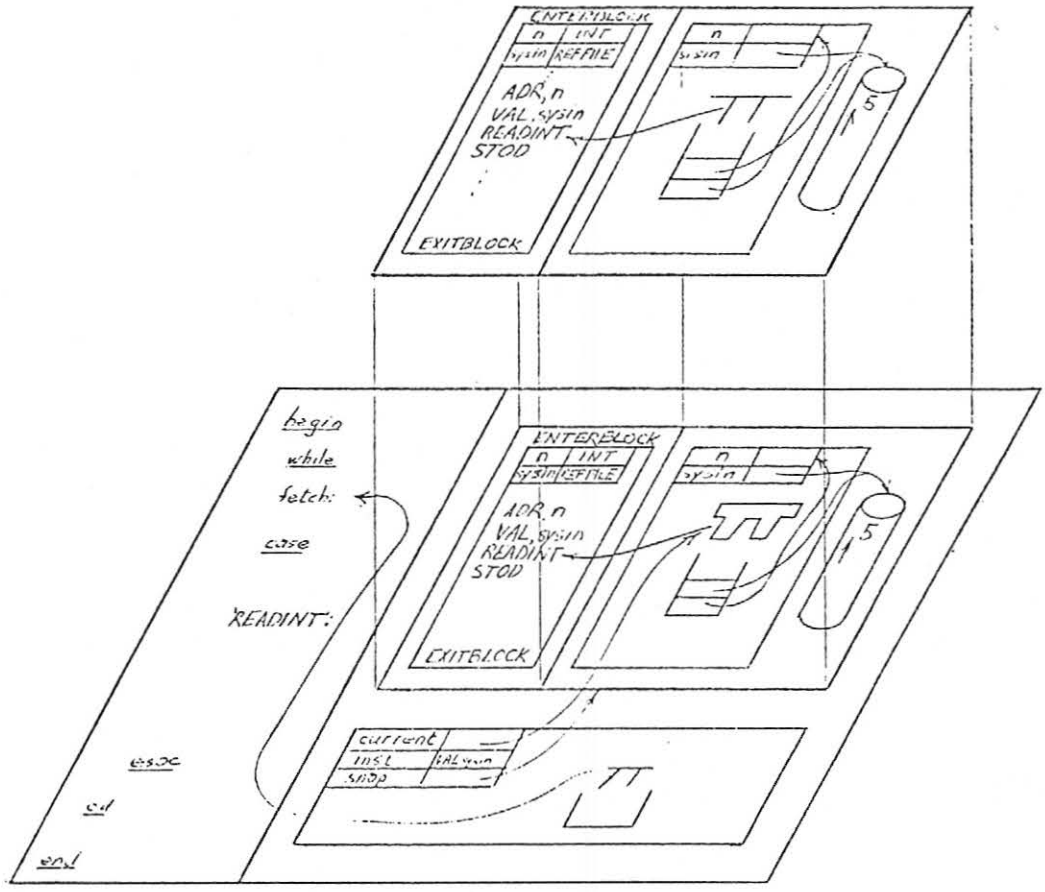
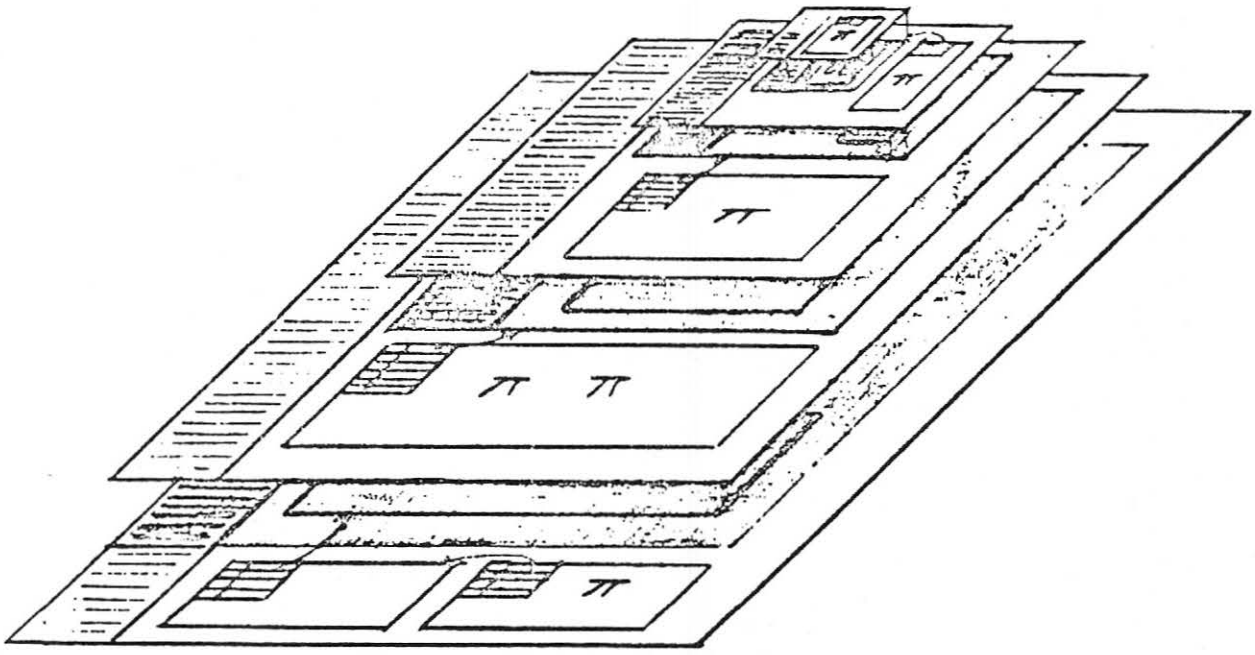


Figure 8.2



TOWER OF //SYSBABEL
Figure 8.3

geared (old-fashioned) adding machine (the authors are old enough to remember them) where the lowest level is the unit's gear, the highest level is the billions gear, and the lowest level gear must move 10^9 teeth to move the highest level gear one tooth.

9. Conclusion

We have attempted to characterize the multilevel system phenomenon from an Information Structure Model point of view. We first identified several methods of supporting one machine by another. We then gave an Information Structure Model framework for considering a method of support as a pair of models, one implementing and one implemented, together with two mappings, a loading and a simulation map, between them. Then three of the identified methods of support were described in terms of the two models and the two mappings. We concluded with an overall view of a multilevel system as a tower of models.

10. Bibliography

[Bau73] Bauer, F. L. (ed)., Advanced Course on Software Engineering, Berlin: Springer Verlag (1973).

[Bry74a] Berry, D. M., "On the Design and Specification of the Programming Language Oregano," Computer Science Department, UCLA, UCLA-ENG-7388 (1974).

[Bry74b] Berry, D. M., "The Use of Information Structure Models in Programming and Teaching of Programming Languages," Proceedings of Second Jerusalem Conference on Information Technology (August 1974).

[Bry75] Berry, D. M., "Definition of the Contour Model in the Vienna Definition Language," M&M Note #40, Computer Science Dept., UCLA (October 1975).

[BEJS77] Berry, D. M., M. Erlinger, J. B. Johnston, A. von Staa, "Models of Hierarchical Machine Support: Interpretation, Enmasterization, Virtualization, Software Extension, and Compilation," IM #155, Computer Science Dept., UCLA (1977).

[Bri70] Brinch Hansen, P., "The Nucleus of a Multi-Programming System," CACM 13:4 (April 1970).

[CDMPS73] Chirica, L.M., T. A. Dreisbach, D. F. Martin, J. G. Peetz, and A. Sorkin, "Two PARALLEL Euler Run Time Models: The Dangling Reference, Imposter Environment and Label Problems," Proceedings of ACM Symposium on High Level Language Computer Architecture, SIGPLAN Notices 8:11 (1973).

[Den73] Dennis, J. B., "The Design and Construction of Software Systems," in Bau73 (1973).

[Dij68] Dijkstra, E. W., "The Structure of 'THE' Multiprogramming System," CACM 11:5 (May, 1968).

[Goo73] Goos, G., "Hierarchies," in Bau73 (1973).

[Joh69a] Johnston, J. B., "Structure of Multiple Activity Algorithms," Proceedings Third Annual Princeton Conference on Information Sciences and Systems (1969).

[Joh69b] Johnston, J. B., "Structure of Multiple Activity Algorithms," Proceedings of Second ACM Symposium on Operating Systems Principles (1969).

[Joh71] Johnston, J. B., "The Contour Model of Block Structured Processes," in TW71.

[Joh73] Johnston, J. B., "Identifier and Environment Bindings in Nested Declaration Computations," Proceedings of Seventh Annual Princeton Conference On Information Science and Systems (1973).

[JBM74] Johnston, J. B., D. M. Berry, and D. P. Murphy, "Expression Stack Management in Nested Declaration Computations," Proceedings Eighth Annual Princeton Conference on Information Sciences and Systems (1974).

[Joh75] Johnston, J. B. "A Model of the Connective Structure of Segmented Virtual Storage Systems," NMSU-CS-TR-75-01, Computer Science Dept., New Mexico State University (January 1975).

[LZ74] Liskov, B. H. and S.N. Zilles, "Programming with Abstract Data Types," SIGPLAN Notices 18:11 (April 1974).

[Man75] Manthey, M. J., "Nested Interpreters and System Structure," Matematisk Institute, Aarhus Universetet (September 1975).

[Org73] Organick, E. I., Computer System Organization: The B5700/B6700 Series, New York: Academic Press (1973).

[OFP78] Organick, E. I., A. I. Forsythe, and R. P. Plummer, Programming Language Structures, New York: Academic Press, in press.

[TW71] Tou, J. T., and P. Wegner (Eds.), Proceedings of ACM Conference on Data Structures in Programming Languages, SIGPLAN Notices 6:2 (February 1971).

[Weg71] Wegner, P., "Data Structure Models for Programming Languages," in TW71.

[ZR68] Zurcher, F. W. and B. Randall, "Iterative Multi-Level Modelling - A Methodology for Computer System Design," IFIP Congress '68 (August 1968).

[Sam69] Sammet, J. E. Programming Languages: History and Fundamentals, Englewood Cliffs: Prentice Hall (1969)

[MLB76] Marcotty, M., H. F. Ledgard, G. V. Bochmann, "A Sampler of Formal Definition", C. Surv. 8:2 (June 1976).

[Bib??] Hertz, H. H. (Ed.), The Pentateuch and Haftorahs (Second Ed.) London: Soncino (1964).

[JW74] Jensen, K., and N. Wirth, PASCAL User Manual and Report, Berlin: Springer Verlag (1974)

DISCUSSION

Andrei Ershov: Is your model observational or implementational?

Berry: Observational.

Jack Dennis: You have given us some tools and some descriptive models. What should we learn from your work?

Berry: What we have gained is a clearer understanding of what actually happens in a system. For instance, we have clarified the difference between a process and a processor at a given level.

Dennis: I did some thinking some years ago about hierarchical models, and was led to the conclusion that the fewer level, the better, because the user program which is executing at the outermost level depends on the correctness of all of the levels below it. If you are interested in simplicity, and confidence that systems work correctly, it seems that you should reduce the number of levels.

Berry: I agree we should try to keep the number of levels down. The purpose of what I present is to show what exists. Perhaps by understanding what exists, we can see what should exist.

Lawrence Flon: I don't understand why many levels cause difficulty, because the program is correct if it can be shown to operate correctly given that the topmost level satisfies its specifications.

Dennis: You are correct. But the more complex and elaborate the implementation is, the more likely it is that the implementation does not reflect its specifications. So one worries about the confidence the user of an outer level machine has that the machine meets its specifications. My plea is that the underlying hardware be much more accommodating to the program structure and methodology desireable at the user level. Then the overall structure of the system will be simpler, increasing user confidence in its correct operation.

Malcolm Newey: Jack seems to be arguing against modularity. He proposes pushing all the levels into one level.

Dennis: That hurts me very much, of course. The whole machine should support modularity of programming at the outermost level. To me, modularity is the ability to take programs that have been written and use them as building blocks to build other programs which in turn become new building blocks, etc. There is no such relationship in the levels of an onion.

Berry: In a sense, a module may be thought of as presenting a machine, and the act of composing a higher level module as construction of a higher level machine (as described in the models in the paper). This seems to be what he (Newey) is implying by his comment.

Dennis: If you regard the onion as representing a modular scheme for building larger elements out of simpler ones, then I challenge you to take two levels of the onion and define some sensible way of combining them to form a new level.

Berry: Yes, they're not composable. Maybe that's why nobody likes these systems.

FORMAL DESCRIPTION OF PROGRAMMING CONCEPTS

Proceedings of the IFIP Working Conference on
Formal Description of Programming Concepts
St. Andrews, N. B., Canada, August 1-5, 1977

edited by

ERICH J. NEUHOLD
University of Stuttgart,
Stuttgart, Germany

1978

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM • NEW YORK • OXFORD

IFIP TC-2 Working Conference on
Formal Description of Programming Concepts
St. Andrews, N. B., Canada, August 1–5, 1977

Organized by
IFIP Technical Committee 2
Programming
International Federation for Information Processing

Program Committee
H, Bekic, J. deBakker, A. Ershov, M. Hammer, T. Hoare
S. Igarashi, R. Milner, M. Paul, C. Pair

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM • NEW YORK • OXFORD