AN ENGINEERING FORMALIZATION OF COMPUTER SYSTEMS

Thesis by

Luis Manuel Medina-Vaillard

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1978

(Submitted July 25, 1977)

ACKNOWLEDGMENTS

I wish to extend my grateful thanks to Derek H. Fender and his Vision Group for providing an exciting and challenging research environment together with encouragement and support.

The work reported in this Thesis was developed under a fellowship of the Consejo Nacional de Ciencia y Tecnologia of the Government of Mexico.

ABSTRACT

The aim of this work is to develop a formal apparatus that defines the elements composing software-hardware computer systems and the relations between them.

The basic tools come from Systems Theory, Mathematical Logic, Operating Systems and Computer Architecture. Two sound engineering principles are fundamental in the formal apparatus: hierarchical organization and elimination of time dependent malfunctions (like race conditions in hardware and mutual exclusion in software).

All the basic elements in a computer system are identified as 'singletons.' Typical singletons are: (sequential) user programs, (concurrent) operating systems, virtual machines and hardware machines. A computer system is just a set of inter-related singletons.

Velocity and transparency are defined and used as a measure of the quality of a hierarchy. Computer systems are analyzed as directed graphs (singletons being the nodes), and directions for research in optimal structuring are suggested. Singletons are analyzed as a hierarchy of spaces (low level, high level, assertions) and the mappings between them (compiler, programmer).

Finally, a real time, special-purpose multiprocessor system (ECOS) is described and analyzed within this formal framework.

TABLE OF CONTENTS

I. INTRODUCTION

Modern computer systems are complex engineering products, each of which vaguely resembles a von Neumann machine. The new machines are hardware-software structures which pose new problems and constraints like multiprocessing, parallel-processing, storage hierarchies, input/output management, real time processing, pipelining, and above all, an unprecedented number of components. The formal structures of computability theory (like Turing Machines and Markov Algorithms) do not provide appropriate concepts to deal with these new problems, therefore new general tools are constantly being developed. Examples of such successful new general approaches are: "Process Structuring" (semi-formal approach to software concepts[H1], some ideas are similar to those developed in this thesis), CDL (Computer Design Language[C1] - low level register transfer hardware design), PMS and ISP (Processors, Memories, Switches, and Instruction Set Processor[B1] - high level hardware design), and Concurrent Pascal[B4] (operating systems design).

However, as far as I know, all the new general approaches concentrate on either hardware or software aspects of computer systems; so, even though researchers are fully aware that complete systems have to integrate hardware and

software, they usually treat them separately. Nevertheless, there have been specific projects that deal with software and hardware as a coherent set; these projects can be grouped together under the name of high level language computer architectures[C2]. In fact, some of these high level language architectures are very successful engineering products; however, they do not provide the general concepts that could be used in the analysis and design of other computer systems.

In this thesis I will develop a formal structure which can be used to describe and to design general computer systems based on the sound principle of hierarchical structuring.

Much has been said in favor of hierarchical structures; for example Simon[S1] puts hierarchies at the center of his discussion of "the science of design" and the "architecture of complexity" because hierarchies simplify the design task by partitioning a large problem into smaller subproblems. A second important property of hierarchies that Simon discusses is that they provide a high probability of a successful evolution for large systems, and this is because a hierarchic system is composed of a small number of stable intermediate subsystems instead of a large number of little pieces. As a third important property, Simon mentions the understandability of hierarchical systems because the

hierarchy itself provides a natural framework for the analysis of the system, and this framework allows the (near) decomposability of the system but preserves the relevant information for the analysis.

Another classic reference about structuring in computers is Dijkstra's "notes on structured programming"[D2], there, it is demonstrated that structuring is essential in the design, development, modification and understanding of programs.

Here, I will not argue any more in favor of the principle of hierarchical organization and will accept its soundness. This does not mean that any hierarchical system is best under every circumstance; poor hierarchization could mean increased complexity and speed degradation. These two drawbacks will be considered and measured within our formal structure.

A second important principle that will be incorporated in our structure is a simplified concept of time. Two examples of the relevance of this principle are synchronous against asynchronous digital logic, and the monitor concept in concurrent programming (Appendix B). In essence, what a synchronous design does is to transform a continuous time into a discrete one, and this hides the adverse effects of propagation delays. The monitor concept accomplishes a

very subtle transformation of time. Parallel programming with monitors becomes a set of loosely connected sequential programs.

In my structure, time will depend on the characteristics of each subsystem (as in an event driven simulation), and furthermore, within each component, time will always be sequential (as we humans perceive it in the real world).

The tools that I will be using come mainly from Systems Theory[M2,M3,Z1], although I will often refer to concepts from Mathematical Logic[M1], Operating Systems[B2,B3], Computer architecture[B1] and a certain amount of common sense. Fig I.1 represents graphically (without scaling!) the scope of this work.

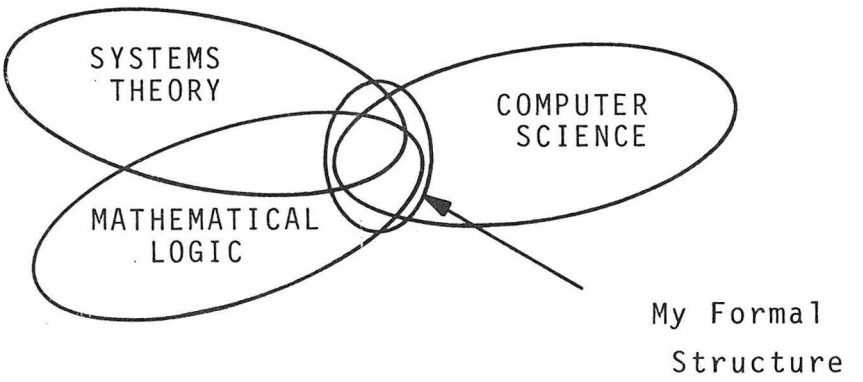


Fig. I. 1 Scope of this work.

It has been said[B1] that: "A digital computer is in principle, representable as a state system, but the number of

states is far too large to make it useful to do so." This is true if the representation is algebraic and with the purpose of obtaining a numerical equation or a transition diagram that explains (!!) the machine. However, it is probably the best representation if the purpose is to obtain a deep understanding of the basic principles of a computer system[D3]. In this thesis I shall give arguments that support this last view.

My formal ideas were developed and tested during the design and implementation of a computer system called 'Experiment Controller System' (ECOS). This special purpose system is tailored to the needs of psychophysical and neuro-physiological experimentation, i.e. experiments in which physical processes constitute stimuli to a living organism, and the behavior of the organism in response to such stimuli constitutes the outcome of the experiment. The ECOS system is relevant to this thesis not only from a historical point of view, but it also provides an example of the use of the formal framework.

The structure of my presentation is as follows: Chapter II gives an informal, introductory view of my structure; Chapter III formalizes all the concepts; Chapter IV elaborates on the applications and implications of this work, and Chapter V applies my formal structure to the design of the ECOS computer system.

The appendixes provide introductory examples for

the reader unfamiliar with the state space approach in systems theory (Appendix A), or with concurrent programming tools (Appendix B).

Before proceeding with the core of my work, I should make two important observations:

- This is not an attempt to develop a new programming language. Instead I will develop concepts that, if important, could be incorporated into new languages.

- The disciplines of Computer Architecture and Operating Systems will be studied together under the name of Computer Systems.

II. BASIC CONCEPTS

As an introductory comment, I should point out that this chapter is an informal discussion of my approach to the central concepts of computer systems. My purpose at this point is only to motivate and to introduce the formalization that will be presented in Chapter III. In this chapter, some of the ideas that will be introduced will have loose ends, but all of them are cleared up in the next chapter where the structure is formalized.

Modern computer systems[B2, B3, D1, T1] are functionally hierarchical[M2]; the user program runs on top of an operating system, the operating system runs on top of a virtual machine, the virtual machine runs on top of a firmware machine, and the firmware machine runs on top of a hard-wired machine. Notice that the hierarchy that I am refering to is a functional hierarchy, it is not a language hierarchy. The elements of this hierarchy (machines) have several things in common, namely:

- Each one exists within an environment defined at a lower level in the hierarchy (except the lowest level whose environment is defined in an 'axiomatic way'*).

---

*In practice, most of the systems do not have a properly defined set of axioms, instead they only have a few, imprecise user's guidelines.

- Each one defines a new environment, or environments. that will support one or several higher machines (except the highest whose function is to interact with an external environment).

- Each level has two kinds of components, passive (data structures), and active (operations on data).

- Each machine, by itself, is capable of executing only one action at a time, i.e., each machine moves sequentially within its environment.

In fact, if we ignore the internal idiosyncrasies of each machine, then the only difference between them is the position that each one occupies in the hierarchy. Considering this, it is very reasonable to conceive a general definition of computer systems as a set of one basic type of element.

The basic building block in my definition will be the singleton. A singleton is a system that possesses all the previously mentioned properties, namely: it exists within an environment defined by another system; it defines a new environment to support a new system; it is composed of active and passive elements; and, it operates sequentially on its environment. Such a system is represented in Fig. II.1.

A computer system is just a set of hierarchically related singletons.

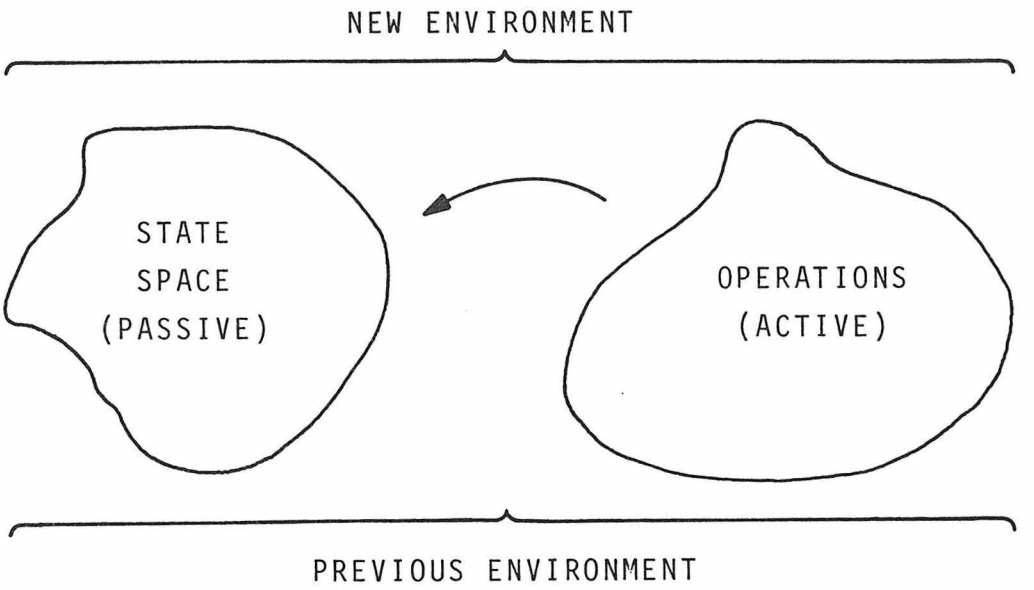


Fig. II.1. Singleton

We are now going to explore these concepts more explicitly by considering an operating system written in Concurrent Pascal[B3,B4]. The reader unfamiliar with concurrent programming tools should refer to Appendix B before proceeding with the rest of this chapter.

The axioms that define the lowest singleton are of two kinds: passive: the store and the registers of the hardware machine; active: the instruction set of the hardware machine. As far as the hardware machine is concerned, it can only be in one state, namely: power on. (The power off condition is totally out of the control of the machine, and totally uninteresting.)

When the hardware machine is in the power on state, it provides an environment in which other machines can exist. This environment consists of a state space and a state transition function (Appendix B). The new singleton that exists in this new environment is usually called the virtual machine.

If a singleton A defines the environment for a singleton B, then A is called the master of B, and B is called the slave of A. The hardware machine is the master of the virtual machine, and this is the slave of the hardware machine.

The virtual machine itself is composed of active components (statements in a programming language) that operate on passive components (data structures); the passive components are mapped on the state space provided by the environment; the active components use the state transition function to operate on the data structures. Operations on the data structures imply changes in the state of the virtual machine, so a sequence of operations implies a sequence of states. A sequence of operations is called a run, a sequence of states is a trajectory in the state space; when the virtual machine runs, it generates a trajectory in its state space.

Notice that the virtual machine can travel an arbitrarily long trajectory within its state space without the hardware machine changing state; this will rarely be the case with other singletons, however, the optimal environment

that a singleton can have is the one in which the singleton can move within its state space without its master changing state, this is because as far as the slave singleton is concerned, any change in the state of the master is overhead (this will become more clear when we examine the topmost singletons in the hierarchy).

By running on the environment provided by the hardware machine, the virtual machine defines a new environment for the next singleton; that is, the definition of the new environment is based on the previous environment as Fig. II.1 illustrates.

On top of the Virtual Machine we have a Concurrent Pascal program, which consists of active components (processes) operating on passive components (monitors*) and defining one or several new environments.

Under this framework, we should realize that the so-called 'mutual exclusion problem' is non-existent. To execute a program means to describe a sequential path within state space, but to change state means to transform the passive components, i.e., to operate on

---

* - At this point we should emphasize the fact that monitors are not active components. They are as active as a real number can be; in fact, the only difference between say an integer and any typical monitor (besides exclusive access which is irrelevant at this point) is that the operations defined on an integer are 'automatically' provided by the environment, while the operations defined on the monitor are defined by the programmer. Conceptually, the difference is minimal, with respect to programming the difference is gigantic.

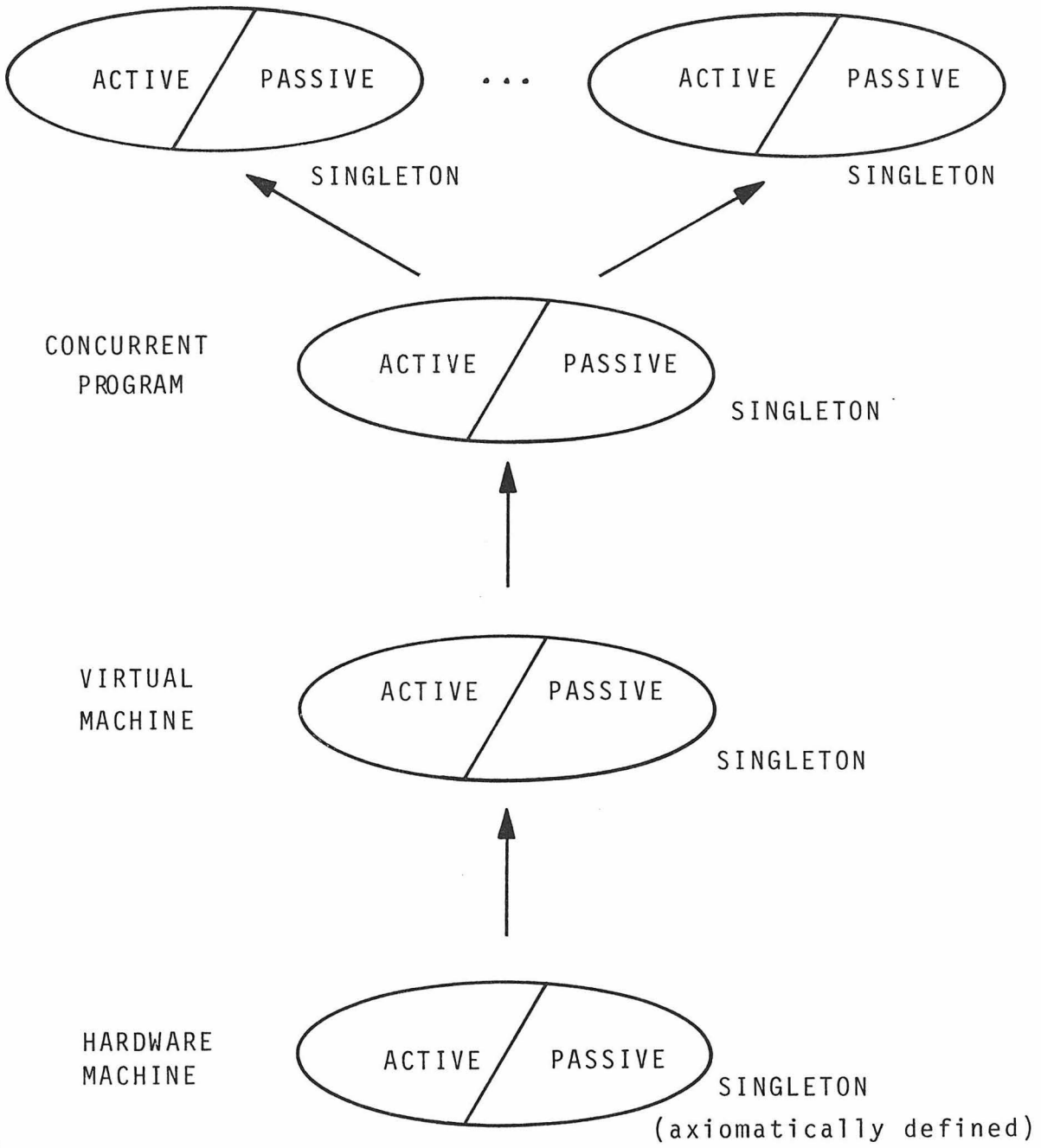


Fig. II. 2 - A hierarchical computer system.

the monitors. If there is no change in the passive components, there is no change in state; similarly, if processes do not operate on a monitor, there is no change of state. Since state transitions are indivisible operations, only one process at a time can operate on each monitor (more about this in the next two chapters).

In general, a concurrent program will define several environments, each of them will provide the state space needed by the next level of singletons, called sequential programs.

Each sequential program consists of active components (statements) and passive components (data structures); sequential programs are the topmost singletons, and their function is to describe a path in the user's assertion space.

Fig. II.2 provides a graphical representation of all the singletons and their relationship, the arrows indicate the master-slave relationship. The level of a singleton is the number of singletons required to define its state space starting from the basic axioms. The level of the hardware machine is zero, and the level of each sequential program is three. A set of singletons of the same level defines an echelon.

In this particular example (Fig. II.2), the echelon level zero has only one singleton, and it is the hardware

machine; other computer systems of interest will have several hardware machines running on top of an abstract singleton (so-called multiprocessor systems[E1]), and still others will have several singletons at level zero (so-called computer networks[R1]).

Notice that when a sequential program runs, it may or may not require changes in the state of its master; the master will change state when the user program requires services that are not provided by its own environment, i.e. if the state transition function of the slave is partial within the slave's context, then it has to be computed by the master of the slave, and this computation requires changes in the state of the master. While the master is doing such computation, the slave (sequential program) is idle, and that is why the optimal environment for a singleton is the one that does not require changes in the state of the master.

Notice also that since the sequential programs are the ones that interact with the environment external to the computer system, they are the ones that accomplish the useful work (as far as the external world is concerned).

As a closing remark, let me emphasize that the hierarchy of singletons is not a hierarchy of languages, instead, it is a hierarchy of state spaces and state transition functions.

III. THE FORMAL STRUCTURE

My formal apparatus is based in the concepts of system and state space representation as approached by Mesarovic and Takahara[M3]; for completeness I will repeat here some of their basic concepts, however since our systems of interest are computers, I will deal with general systems rather than with input-output systems; furthermore, some of my definitions even though directly inspired by[M3] contain significant differences. The reader who is not very familiar with state spaces should read this chapter quickly, then study Appendix A, and then come back to this chapter.

- Definition III.1 SYSTEM[M3]. A <u>general system</u> S is defined as:

$$S \subset x \left\{ V_i : i \in I \right\}$$

where I is an index set of any cardinality, the

components V_i are termed system's objects.

- Definition III.2. GLOBAL STATE RELATION. Let S be a general system; if C' is a set and R a relation R: C' x S such that:

$$s \in S \Leftrightarrow \exists c \in C' \ (R(c,s))$$

then C' will be called a <u>global state object</u>, an element of C' a <u>global state</u>, and R will be called a <u>global state relation</u>.

Notice that a global state relation provides a more specific view of a general system, since by giving a global state it picks out distinguished elements of S. However, we still need additional structure since computer systems are constantly picking out different elements of S. This behavior requires not only the set S, but also a set that will impose an order on the elements being selected out of S. This set will be called time.

- Definition III.3. TIME. A set T will be called time iff it is a well ordered set. i.e., there exists a relation

'$<$' on T, such that:

i) $\vdash \forall a \in T \quad (a \not< a)$

ii) $\vdash \forall a \forall b \forall c \in T \quad (a<b \wedge b<c \Rightarrow a<c)$

iii) $\vdash \forall a \forall b \in T \quad (a<b \vee a=b \vee b<a)$

(i to iii define linear order),

iv) $\vdash \forall X \subseteq T \quad (X \neq \emptyset \Rightarrow \exists x \in X \ (\forall y \in X \Rightarrow x<y \vee x=y))$

(i.e. every subset has a least element)

Well ordering is required, not just ordering, so that any time segment will always have an specific 'beginning'.

As a convention the least element of a time set T will be denoted by t_0. The 'succ' and 'pred' functions will have their usual meaning and we will denote: $t_1 = \text{succ}\ (t_0)$, $t_2 = \text{succ}\ (t_1)$, $\cdots$ and $\cdots t_1 = \text{pred}\ (t_2)$, $t_0 = \text{pred}\ (t_1)$; in general $t^+ = \text{succ}\ (t)$, $t^- = \text{pred}\ (t)$.

- Definition III.4. TIME FUNCTION. A <u>time function</u> f from a set $\mathscr{F}$ is a mapping from a time set T into $\mathscr{F}$. Given $t \in T$, we will denote the unique value of f at t as f(t), and will call it a <u>point</u> of f. The set T will be called the <u>time base of f</u>.

- Definition III.5. DYNAMIC OBJECT[M3]. A dynamic object F is a set of time functions from $\mathscr{F}$. The set $\mathscr{F}$ is called the alphabet of F.

- Definition III.6. TIME SYSTEM. A system $S \subset X\{V_i : i \in I\}$ will be called a time system with time base T iff $\forall i \in I$ V_i is a dynamic object with time base T.

The elements of a time system are n-tuples of time functions, and considering this, we can rewrite definition III.2 (global state relation) so that it reflects the dynamic property of a time system.

- Definition III.2'. GLOBAL STATE RELATION. Let S be a time system with time base T, $t \in T$, $s \in S$. Furthermore, let C' be a set, $\mathscr{R}$ the family of all relations R of the form R: C' x s, such that:

$$s \in S \Leftrightarrow \exists R \in \mathscr{R} \, (\forall t \in T \forall c \in C' \; (R(c, s(t))))$$

(compare with Definition III.2)

then C' will be called a global state object, an element of C' will be called a global state, $\mathscr{R}$ will be called a global state relation, an element of $\mathscr{R}$ will be called a dynamic global state relation.

At this point, I can define two of the basic concepts to be used in the definition of computer systems, namely: state space and state transition function.

- Definition III.7. STATE TRANSITION FUNCTION. Let S be a time system, T its time base, C′ its global state object, $\mathscr{R}$ its global state relation; t, $t^{+} \in T$. Then let

$$S_t = \{x(t) : x(t) \in s \wedge s \in S\}$$

be the expansion of S; i.e. S_t is the set of all the points of all time n-tuples in S.

Let us define a function φ: $C' \times S_t \rightarrow C'$

such that:

$$\forall s \in S \forall R \in \mathscr{R} \forall t \in T \forall c, c' \in C'$$

$$(\varphi(c, s(t)) = c' \Leftrightarrow R(c, s(t)) \wedge R(c', s(t^{+})))$$

(evolution in time of system's state)

then φ will be called the <u>state transition function of S under C′</u>.

- Definition III.8. STATE SPACE. Let S, T, C′, S_t be as in III.7; then let φ be the state transition function of S under C′. Define a relation E : C′ x C′ as:

$$E = \{(c, c') : \forall x \in S_t \ (\varphi(c, x) = \varphi(c', x))\}$$

Clearly, E is an equivalence relation, and I will denote [c] the equivalent class of c under E. The <u>state space</u> C of S is defined as:

$$C = \{[c] : c \in C'\}$$

Since C is unique up to isomorphism, φ under C will be called the state transition function of S.

- Definition III.9. SPACE TRAJECTORY. Let S be a time system, T its time base, C its state space, φ its state transition function. A space trajectory Φ defined by $x \epsilon S$ is a time function with base T denoted by $\Phi_{c0,x}: T \longrightarrow C$ such that:

$$\Phi_{c0,x}(t) = \begin{cases} c0 & \text{when } t = t_0 \\ \varphi(\Phi_{c0,x}(t^-),\ x(t^-)) & \text{when } t > t_0 \end{cases}$$

The state c0 is called the initial state of S. The sub-index co,x emphasizes that the trajectory is totally determined by the initial state and the time function. Informally, we will extend the functions pred and succ defined on T, to the set of states $\{c0,\ c1,\ c2,\ \cdots\}$.

In practice, sets are usually defined by characteristic functions, i.e. functions that applied to a given item tell if it is a member of the set or not. State spaces are no exception, and within our context they will often be defined by a set of variables (a vector), then an assignment to all the variables will define a point in the space. Therefore, if two spaces intersect, that means that they share one or more variables in their definitions, and

since the variables cannot have two different values simultaneously, then points that lie in the intersection can only exist one at a time. These ideas will be very important in the definition of multiple slaves.

With all the previous fundamental concepts I can now start defining the basic components of a computer system.

- Definition III.10. PROGRAM. Let S_p be a time system with finite index set I, and state space C; let $\{I_s, I_d\}$ be a partition of I. Let us define:

$$S = x \{V_i : i \epsilon I_s\}$$

$$D = x \{V_i : i \epsilon I_d\}$$

$$p0 \epsilon C$$

Consider $s \epsilon S$, $d \epsilon D$; a <u>program P from S_p</u> is defined as:

$$P = \{s, d, p0\}$$

The system S_p is called a <u>class of programs</u>; S is called the <u>class of statements</u> of S_p; D is called the <u>class of data</u> of S_p; s is called the <u>statements</u> of P; d is called the <u>data</u> of P; p0 is called the <u>initial state</u> of P.

An essential part of a program is its initial state, this fully agrees with the 'initialization of variables' suggested in [D3].

- Definition III.11. ENVIRONMENT. Let S_p be a class of programs. Let E_c be the state space of S_p, and E_φ be the state transition function of S_p. Then a set E defined as:

$$E = \left\{ E_c,\ E_\varphi \right\}$$

will be called the environment of S_p.

- Definition III.12. SINGLETON. Let S_p be a class of programs, T its time base, $E = \left\{ E_c,\ E_\varphi \right\}$ its environment, $P = \left\{ s,\ d,\ p0 \right\}$ a program from S_p, denote by $\Phi_P = \Phi_{p0,\{s,d\}}$ the space trajectory of P in E_c. Then the set:

$$S = \left\{ E_c,\ E_\varphi,\ P \right\}$$

will be called a <u>singleton</u> with time base T.

In short, a singleton is an environment and a program.

- Definition III.13. MASTER-SLAVE RELATIONSHIP. Let

$$S_1 = \left\{ E_{c1},\ E_{\varphi 1},\ P_1 \right\}$$

$$S_2 = \left\{ E_{c2},\ E_{\varphi 2},\ P_2 \right\}$$

then S_1 will be called the <u>master</u> of S_2, and S_2 will be called <u>slave</u> of S_1 iff:

$$\text{i)} \qquad \forall c \epsilon E_{c2}\ \exists x \epsilon E_{c1}\ (c \epsilon x)$$

(the slave's space is defined by the master)

ii) $\forall x \in E_{c1} \exists c \in x \ (c \in \Phi_{P_2} \Longrightarrow \exists c_t \in x \wedge$

$c_t \in \varphi_1(x, P_1) \wedge$

$(\varphi_2(c_t, P_2) = \varphi_2(\varphi_1(x, p_1)_t, P_2)))$

c_t is called an intermediate state.

(Master's states contain intermediate states; when the slave reaches an intermediate state, the transition function is defined by the master.)

Fig. III.1 represents the relationship just defined; the singleton S_1 is master and the singleton S_2 is slave. Notice that a single point in E_{c1} is a set of points in E_{c2}, notice also that the space trajectory of P_2 is within c0 (as defined by φ_2) until it reaches an intermediate state, at that time φ_2 is computed by means of φ_1, this implies that P_1 moves from c0 to c1, and after that the trajectory of P_2 will be within c_1. The master does not tell the slave what to do, it only puts some limitations on what the slave can do.

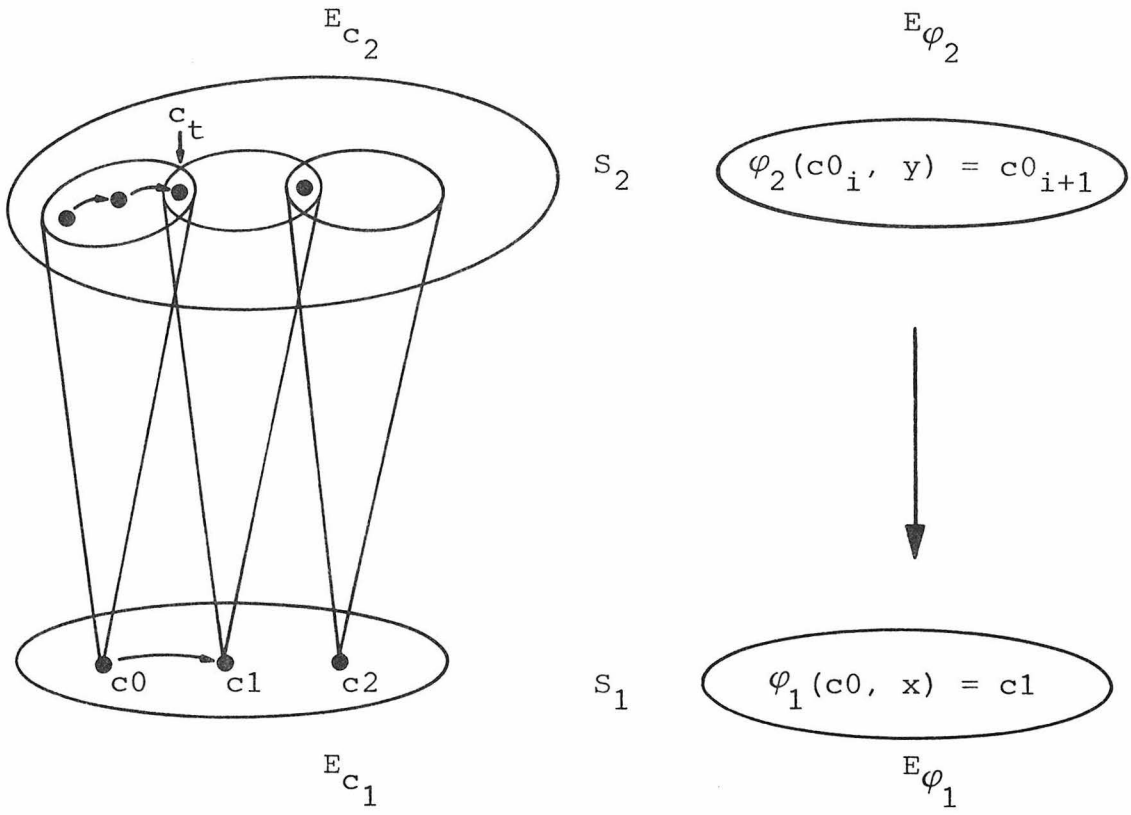


Fig. III.1. Master-Slave Relationship.

From definition III.13 i), we can assume that in general $E_{c2} \neq U\left\{x: x \epsilon E_{c1}\right\}$, and this fact leaves the door open to a master with multiple slaves; however, I should add the following restriction between slaves with a common master.

- Definition III.14. CONSISTENT SLAVES. Let S_1, S_2, $\cdots$ be the set of all singletons which are slaves of S_M.

If $E_{c1} \cap E_{c2} \cap \cdots = \emptyset$

then the set of slaves will be called consistent, otherwise it is said to be inconsistent.

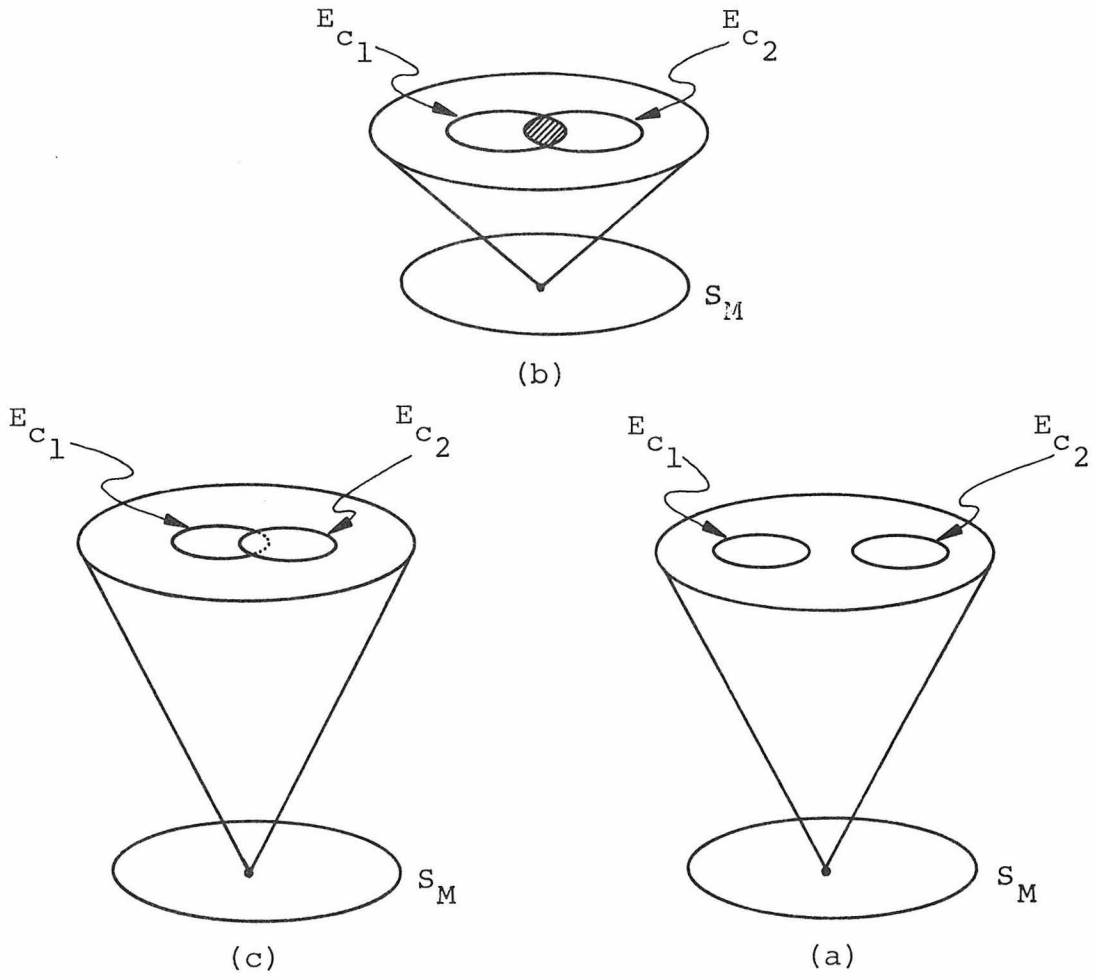


Fig. III.2. Multiple Slaves.

The concept of multiple slaves is illustrated in Fig. III.2; Fig. III.2a shows two consistent slaves, while (b) represents two inconsistent slaves. When the slaves are consistent they can move within their own spaces totally independent of each other. An example of this case in computer systems is the execution of parallel processes. However, when the slaves are inconsistent only one of them at a time can be within the intersection of their spaces; if both of them try to 'execute' inside the intersection at the same time, this means that they both will try to operate on the same variables at the same time, and this can produce random space trajectories.

This is a good place to examine how concurrent programming techniques fit into this picture (Appendix B). When programming with monitors[B2] we have consistent slaves; when one of the slaves wants to operate on monitor variables it enters an intermediate state then the master takes over and does the operations (changes state) and then the slave can continue again.

Brute force parallel programming is equivalent to inconsistent slaves. Programming with semaphores[D4] is represented in Fig. III.2c, they are essentially inconsistent slaves together with a change of state in the master that tells the slaves if anybody is inside the intersection.

In this work we will consider only consistent slaves, and therefore the mutual exclusion problem will be non-existent (by definition) within our formal structure.

- Definition III.15. MULTIPLE MASTERS. The purpose of this definition is to extend the concepts of III.13. Let S_s be a singleton and $S_M = \left\{ S_1, S_2, \cdots \right\}$ be a set of singletons such that:

i) $\quad .- \forall c \in E_{cs} \; \exists S \in S_M \exists x \in S \; (c \in x)$

ii) $\quad .- \forall S_i \in S_M \left\{ \forall x \in S_i \exists c \in X \; [c \in \Phi_{P_s} \Longrightarrow \right.$

$$\Longrightarrow [\exists c_t \in x \wedge c_t \in \varphi_i \; (x, p_i) \wedge$$

$$\wedge \varphi_s \; (c_t, P_s) = \varphi_s \; (\varphi_i (x, P_i)_t, P_s)]_\vee$$

$$\vee [\exists c_\mu \in x \wedge \exists ! \; S_j \in S_M \wedge S_j \neq S_i \wedge$$

$$\wedge \exists y \in S_j \wedge c_\mu \in \varphi_j (y, P_j) \wedge$$

$$\left. \wedge \varphi_s (c_\mu, P_s) = \varphi_s (\varphi_j (y, P_j)_\mu, P_s)] \right\}$$

('can move within one master' or 'can switch masters')

Then we call:

$S_1, S_2, \cdots$: masters of S_s

S_s : slave of $S_1, S_2, \cdots$

c_t : intermediate state within one master

c_u : intermediate state between masters

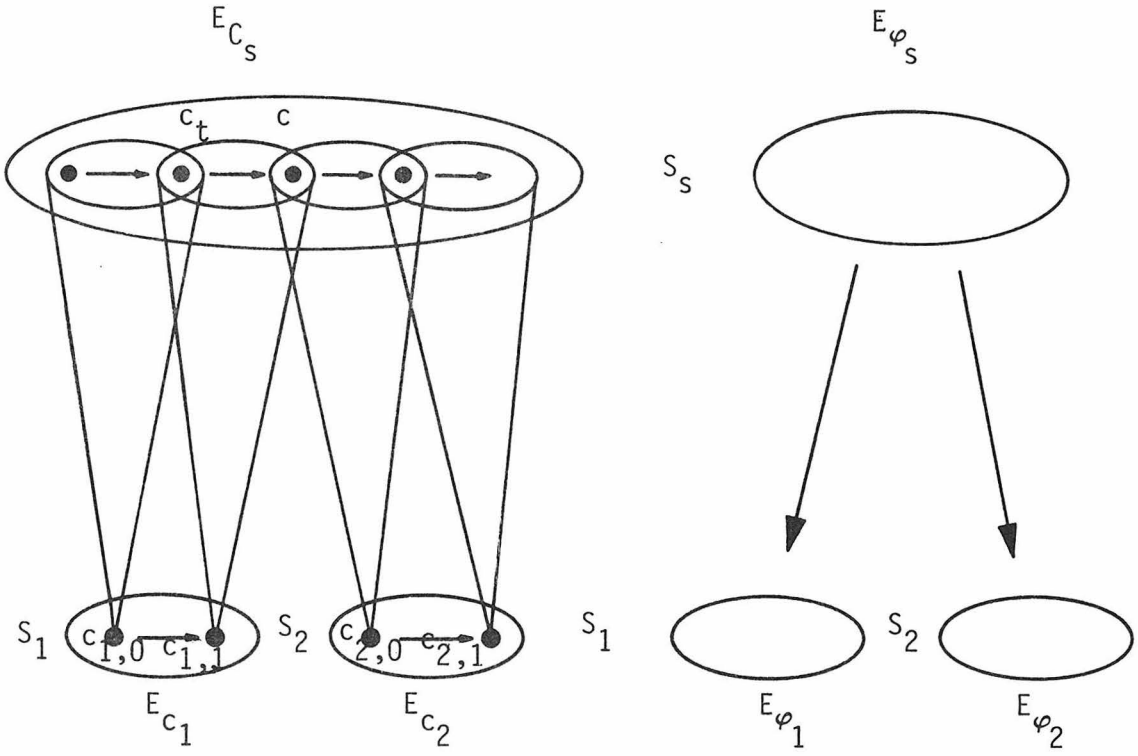


Fig. III.3. Multiple Masters.

The concept of multiple masters is represented in Fig. III.3; S_1 and S_2 are masters of S_s. The slave S_s is moving within $c_{1,0}$ until it reaches an intermediate state within S_1, then S_1 moves from $c_{1,0}$ to $c_{1,1}$ and S_s continues moving within $c_{1,1}$ until it reaches an intermediate state

between S_1 and S_2; at that time S_2 'takes over' S_s by moving to state $c_{2,0}$ and again S_s moves within $c_{2,0}$ until it reaches a transition state within S_2, and the process goes on...

- Definition III.16. GRAPHS[K1]. Notice that we can consider a singleton as a node and the master-slave relationship as a directed edge; therefore, I will define some concepts of graphs that are important in this context.

Two singletons are said to be <u>adjacent</u> iff there is a master-slave relationship between them.

Let us consider a set of singletons $S = \{S_1, S_2, \cdots\}$; a <u>path</u> between two singletons S_i, S_j is a nonempty set $P_h = \{S'_1, S'_2, \cdots S'_k\}$ such that:

i) .- $P_h \subseteq S$

ii) .- $S'_1 = S_i$

iii) .- $\forall\rho(1\leq\rho<k \Longrightarrow S'_\rho$ "is adjacent to" $S'_{\rho+1})$

iv) .- $S'_k = S_j$

k is called the <u>length</u> of the path.

The set S is <u>connected</u> iff there is a path between any two of its elements.

A path $P_h = \{S'_1, S'_2, \cdots S'_k\}$ between two singletons S_i and S_j is called (master) <u>oriented</u> iff:

$$\forall\rho(1\leq\rho<k \Longrightarrow S'_{\rho} \text{ "is master of" } S'_{\rho+1})$$

A singleton which has no slaves will be called a leaf.

(Notice that because of the definitions of singleton and master-slave relationship, there are no paths between a singleton and itself, i.e., there are no 'cycles'. Therefore, if we have a connected set of singletons they form a free tree.)

- Definition III.17. HIERARCHICAL SYSTEMS. A set of singletons S is said to be a hierarchical system iff it is connected. A singleton $M\epsilon S$ is axiomatic iff $\nexists N\epsilon S$ (N "master of" M) (i.e. M is a 'root').

Let $M, N\epsilon S$; M axiomatic; if there is an oriented path between M and N then M is called the absolute master of N and the length of the path is called the level of N with respect to M. If there is no oriented path between M and N, then the level of N with respect to M is undefined.

The level of an axiomatic singleton is zero.

Let $E\subset S$, the set E will be called an echelon of of level k iff:

$$\exists!\ M\epsilon S\ (\forall N\epsilon E\ (\text{level}(N) = k))$$

The set of all axiomatic singletons is called echelon of level zero.

Let us illustrate some of these definitions. Fig. III.4 shows two hierarchical systems, the circles represent singletons and the arrows go from masters to slaves. The system in Fig. III.4a contains four echelons: echelon zero contains only one singleton, which is the absolute master of the whole system; echelons one and two each contains three singletons, echelon three only has one singleton. Echelon three contains one leaf, echelon two, two leaves; and echelon one, one leaf.

Fig. III.4b represents a hierarchical system with three axiomatic singletons, i.e. echelon zero contains three singletons. Singletons 1 and 2 are absolute masters of 4; the level of 4 with respect to 1 is two, with respect to 2 it is also two. Singleton 4 belongs to echelon two with respect to any of its masters. The level of 5 with respect to 1 is undefined, with respect to 2 it is three, and with respect to 3 it is two. So singleton 5 belongs to echelon three when under absolute master 2, and belongs to echelon two when under absolute master 3.

By now we have all the concepts needed to define a computer system, which, as the reader can imagine, is just a special kind of hierarchical system.

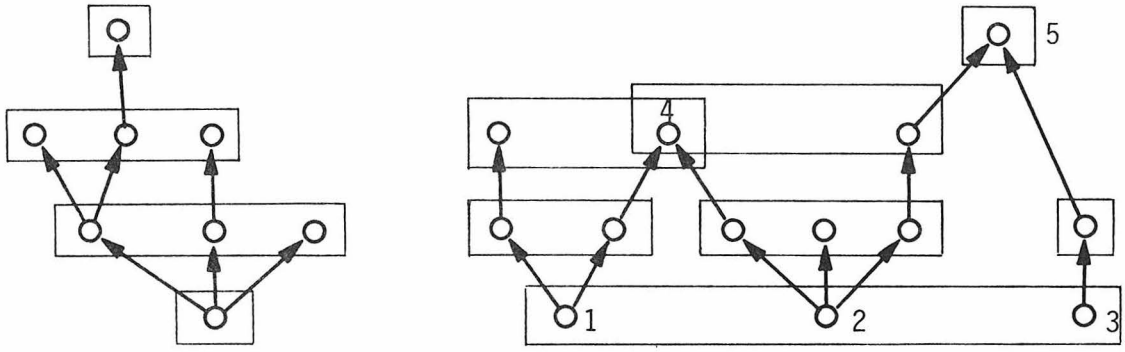


Fig. III.4. Two hierarchical systems.

- Definition III.18. COMPUTER SYSTEM. A hierarchical system S is called a <u>computer system</u> iff:

i) .- its cardinality is finite

ii) .- $\forall x \in S$

(E_{Sx} is finite $\wedge$

$E_{\varphi x}$ is finite $\wedge$

P_x is finite $\wedge$ cardinality (T_x) is at most $\aleph_0$.)

In other words, all the different pieces of the system are finite and the time bases are at most countable infinite.

Once computer systems are well defined, I proceed to define some concepts whose purpose is to quantize important engineering ideas.

- Definition III.19. RELATIVE VELOCITY. Let $S = \left\{ E_S, E_\varphi, P_S \right\}$ be a singleton with Φ_S its space trajectory. Let $g \subset \Phi_S$ such that:

i) .- $g = \left\{ g(1), g(2), \cdots g(k) \right\}$

ii) .- g(1) = succ(c); c is an intermediate state

iii) .- g(k) is an intermediate state

iv) .- $\forall g_i \in g$ ($i \neq k \Longrightarrow g_i$ is not an intermediate state)

then g will be called a segment of Φ_S.

Let G be the set of all segments of Φ_S; the <u>relative velocity</u> of S, denoted as v_r is defined as:

$$v_r(S) = \frac{\sum_{g \in G} card(g)}{card(G)}$$

i.e. it is the average number of states between intermediate states.

- Definition III.20. VELOCITY. Let H be a hierarchical system, let S_k be a singleton, M its absolute master; let $P = \{M, S_1, S_2, \cdots S_{k-1}, S_k\}$ be the oriented path between M and S, then the velocity of S, denoted as v, is defined as:

$$v(S) = \prod_i v_r(S_i) \; ; \; i = 1, 2, \cdots k$$

i.e. it is the average number of states between state changes of the absolute master.

The absolute masters are the slowest singletons of a system; their scope is wider than the slave's. The function of the slaves is to accomplish specific tasks, the function of the master is to supervise the slaves, and therefore the slaves have to do 'something' before they can be supervised!

In the introduction I mentioned two problems of poorly designed hierarchical systems: increased complexity and speed degradation. The concepts of relative velocity and velocity will be used to measure speed variations between different designs; similarly, variations in complexity will be measured using the concept of transparency[P1].

To explain transparency we will use the excellent example of Parnas and Siewiorek[P1]. Consider the car of Fig. III.5a, the front wheels can be oriented with respect to the body of the car, the rear wheels can not. The state space of this system is given by the orientation of the

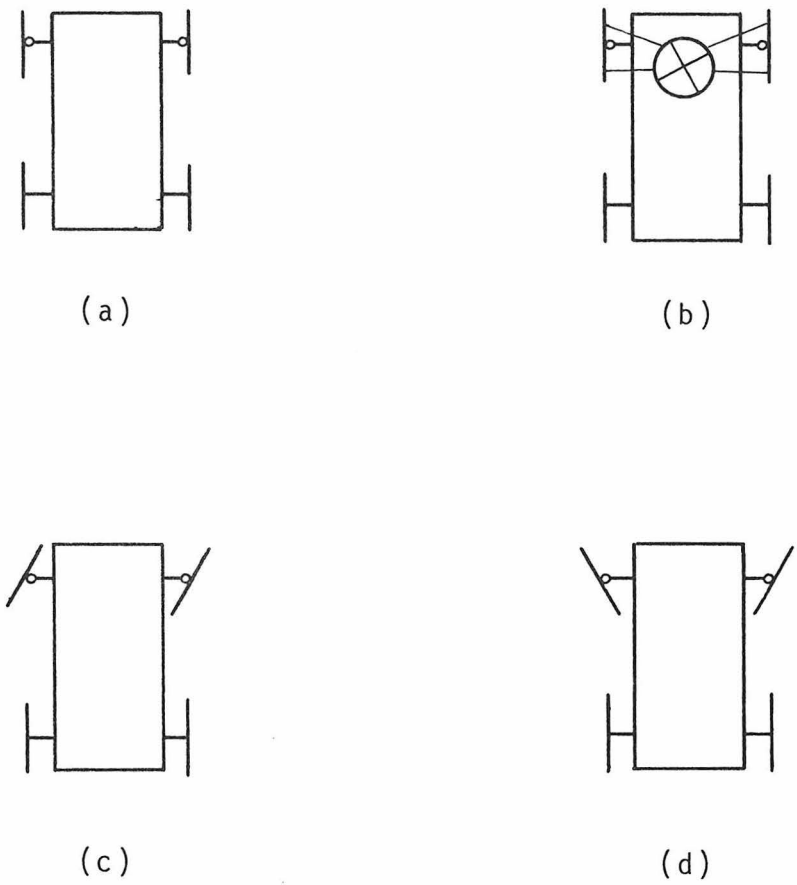


Fig. III.5. The Concept of Transparency.

front wheels. A steering mechanism (Fig. III.5b) implemented by four ropes and a 'steering wheel' defines a subspace of the state space, namely: 'the front wheels are parallel' (Fig. III.5c). This mechanism enormously simplifies the control of the car, and it eliminates 'undesirable states' (Fig. III.5d).

So the abstraction implemented by the steering mechanism limits the number of states that can be reached. Transparency is a measure of the number of states that we can reach from a given abstraction.

- Definition III.21. TRANSPARENCY. Let H be a hierarchical system whose singletons are of the form $S = \left\{E_c, E_\varphi, P\right\}$. Then the <u>total state space</u> is defined as:

$$E^T = \left\{x: x \epsilon E_c \wedge E_c \subset S \wedge S \text{ is a leaf}\right\}$$

i.e. E^T is the total set of lowest level states that the system can reach.

The <u>transparency</u> T of a singleton S is defined as:

$$T(S) = \frac{1}{\log \frac{\text{card}(E^T)}{\text{card}(E_c)}}$$

Then if $E_c = E^T$ the singleton is infinitely transparent, but if we have many echelons, the minimal transparency that we can have is 1/log card (E^T).

Notice that for a singleton to be simple to understand, it should have small transparency; i.e. it should reduce a large number of states into a small number. However, the simplification should be such that all the desirable states are reachable. For instance, going back to the

example of Fig. III. 5, the state represented in Fig. III. 5d was labeled as undesirable, but it could be a desirable state if we want to use it to stop the car; in this last case, the steering mechanism that keeps the front wheels parallel would not be an appropriate design since the state of Fig. III. 5d would not be reachable.

Combining the ideas of velocity and transparency, we can see that a well designed hierarchical system should, first of all, solve the problem that it is intended to solve, and then it should maximize the velocity and minimize the transparency of the complete set of singletons. Notice that my definitions of velocity and transparency apply to a singleton. The concepts of velocity and transparency for a complete hierarchical system have not been formalized in this thesis; the overall system velocity and transparency depend not just on the individual velocities and transparencies of each singleton and the system's graph, but they also depend on the implementation of multiple slaves (IV.2).

It is interesting to relate my formal approach to computability and unsolvability problems. Each singleton can be considered as a Turing machine, thus a computer system can be considered as a hierarchical Turing Machine. Let me look at how a connected set of singletons can treat an undecidable problem. The way to 'solve' an undecidable problem is to add an axiom that makes it decidable. In a batch

computer system the halting problem should be decidable, a user program should always stop in a finite amount of time. The way to solve this problem is to introduce an axiom in the environment of the leaves (the leaves execute the user program) that forces termination of a program after a maximum amount of time. We need not introduce additional enforcement anywhere else to make the overall system halting problem decidable; since we know the configuration of the rest of the system, we can prove decidability anywhere else but in the leaves. What a hierarchical Turing Machine (computer system) does is to restrict the influence of undecidable problems to a small number of singletons, so that by introducing axioms in the environments of those singletons the undecidable problem is decidable in the context of the complete system. Notice that this approach is the same as when dealing with degrees of unsolvability.[M1].

In the following chapters I will explore some applications of this formal apparatus, and I will point out some promising areas for future research.

IV. SEMANTICS OF THE FORMAL STRUCTURE

My purpose in this chapter is to examine how the concepts previously formalized fit into 'standard' computer science structures like compilers, virtual machines, multi-processors systems, etc.

First of all we will look at singletons by themselves, then we will examine sets of slaves and their masters, and finally computer systems will be analyzed.

IV.1 - Representation of a Singleton.

A singleton was defined (III.12) as an environment (state space and transition function) together with a program; nothing was said with respect to its representation.

The simplest, most primitive representation of a singleton is the one that its master can manipulate, this we call a low level representation. A second representation of a singleton is the one that its designer can manipulate, this we call a high level representation. A third representation of a singleton is a synthesis (abstraction) of the purpose (reason of existence) of it, this we call an assertive representation.

Obviously, these three representations are not the only ones that can be constructed, but they are the ones that seem to be more important in practice. Other representations that could be important in practice are: multiple low level representations, so that the singleton can be manipulated by different masters; shortest representation, so that the singleton can be economically stored; fastest representation, the one that maximizes the singleton's velocity, etc.

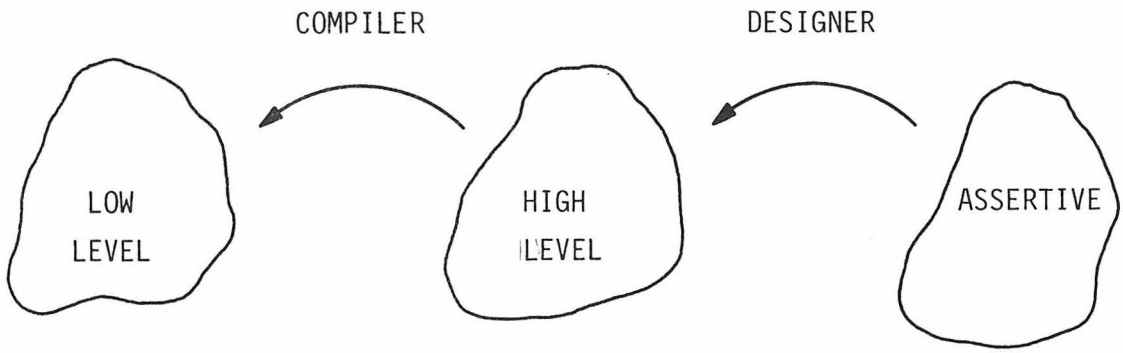


Fig. IV. 1. Representations of a Singleton.

There is no fundamental reason why the representations have to be different. However, at the present state of the art they usually are.

An important practical problem is how to go from an assertive into a low level representation. Fig. IV. 1. illustrates this problem and its solution. The system that transforms the high level into a low level representation is called a compiler. The system that transforms representations from assertive into high level is called a designer

(typically a human system).

The set of rules and symbols that can be used to represent a singleton is called a (computer) language. A language intended for low level representations is called a low level language; a language intended for high level representations is called a high level language, and a language intended for assertive representations is called an assertive language.

The theory of high level languages and compilers is advanced enough that compilers can, and should always verify the syntactic and semantic correctness of the high level representation before attempting any transformation into low level. Let me explain the importance of this verification, remember that the environment of a singleton is given by its master, and the environment determines the class of programs that the singleton can have; if the transformation between high and low level representations is accomplished without previous syntactic and semantic verification, then the singleton's master may not be able to manipulate the low level representation, that is, the low level representation may be outside the class of programs that the environment can execute.

A compiler for a specific high level language is usually constructed with a specific environment as a goal, however, there are already high level languages[B5,W1] and compilers intended for a class of environments rather than

for a single environment. In these cases the high level representation of a singleton is accompanied by a partial description of its environment (a 'prelude'[W1] or a 'prefix'[B5]); it is not hard to imagine (within my formal framework) a high level language and its compiler that could accept a total description of the singleton's environment (Fig. IV. 2).

```
ENVIRONMENT

   TYPE

      byte = ARRAY (.1..8.) of BIT;
      integer = ARRAY (.1..N.) of byte;
      accumulator = ... ;
      index-reg = ... ;
         ...

   VAR   "used by compiler not by user"
      a1, a2: accumulator;
      store: ARRAY (.1..32767.) of UNIVERSAL byte;
         ...

   OPERATIONS
      accumulator:
         ...

   ...
```

Fig. IV. 2. An Environment Description.

Notice that in this discussion there is no need to make any difference as to whether the singleton is implemented by software or hardware, in fact the trend in hardware design is towards a compilation process[B6].

Let me now elaborate on the transformation between the assertive and the high level representations. The task

of the designer is still poorly understood, however among several ad hoc techniques there is a very successful one called structured programming[D2]. Within the context of representation of a singleton, structured programming can be called structured design and it can be explained as an iterative process which generates a sequence of representations of a singleton such that:

- The first representation in the sequence is the assertive representation

- The last representation in the sequence is the high level representation.

- Every two successive representations in the sequence are very similar (similarity being a subjective judgment).

Since successive representations are very similar, transformations between successive representations have to be simple, therefore what structured design does is to partition the original design problem into a sequence of simple designs.

Notice that structured design only simplifies the problem but does not solve it; human judgment, and therefore human mistakes are involved in every step of the design. Then it is very important to ask whether the high level representation is indeed equivalent to the assertive representation. Hardware simulation and correctness proofs

of programs are attempts to answer this question by transforming a high level representation back into an assertive representation.

To end this section, I want to emphasize that we have been looking at the representation of a singleton, which is only a node in the computer system.

IV. 2. Multiple Slaves. Concurrency and Parallelism.

In Chapter III, I mentioned the essential property that a group of slaves must satisfy, I called it consistency (III.14.). In this section I will elaborate on how consistent multiple slaves are usually implemented at use time.

Remember that a master has multiple slaves when it provides several environments, each one for a different program. Nothing has been said with respect to time, since for each slave time is strictly sequential, and for the master time is also sequential. However, in practice we are often interested as to how the slaves move with respect to each other. Fig. IV. 3. illustrates the three different ways in which a group of slaves can move.

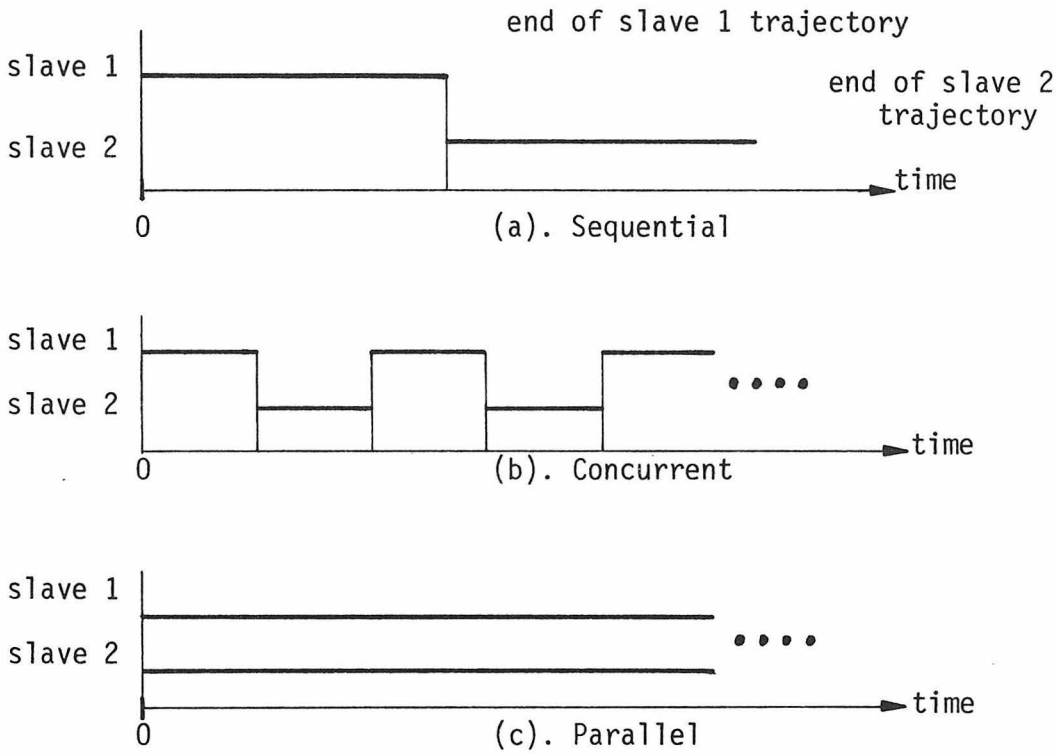


Fig. IV. 3. Multiple Slaves on 'External' Time.

Sequential and concurrent movements are characterized by the fact that only one slave at a time moves. In the sequential case each slave keeps moving until its trajectory terminates; while in the concurrent case they alternate. The sequential movement is of little interest in computer systems since it involves the termination of the singletons. Parallel movement is the case in which all the singletons can move simultaneously. A typical example of concurrent movement is a time sharing system. A multiprocessor system

is an example of parallel movement.

Why should we time the slaves with an external time base if all we need conceptually is that they are consistent? The reason is that we are required to evaluate the performance of the computer system with respect to the external world. My formal structure makes a clear separation between internal structure and performance in the real world. On one side we have the problem of mutual exclusion and a clear design, and on the other side we have 'external' performance. Both are equally important and that is why they require special attention.

The 'practical' difference between concurrent and parallel movements is the total number of actions that can be accomplished in a given amount of external time. In other words, the absolute velocity of the system. In Chapter III, I defined relative velocity (III.19) and velocity (III.20). The absolute velocity of parallel slaves is the sum of the velocity of each slave. The absolute velocity of concurrent slaves is the average of the velocity of the slaves. These are not formal definitions, the absolute velocity of a complex computer system can not be determined by one line recipes. In Chapter VI, I will explore the possibilities of determining specific configurations that maximize the absolute velocity of a computer system.

IV.3. Semantics of the Total System

A computer system is a (finite) connected set of singletons. This means that singletons are defined one on top of the other, except for axiomatic singletons. Therefore, the semantics of the total system is the answer to the question, How do all the singletons map on the axioms?

A demanding reader would also ask: What are the axioms? However, an answer to this depends on how much of the system is going to be designed (i.e. how much is going to be bought and how much is going to be built). Typical sets of axioms are: register transfer modules; memories, processors and switches; complete computers; complete computers and operating systems. Chapter V. describes a system that takes its axioms from all these groups; at this point we are more concerned with the mapping rather than with the axioms themselves.

Let us consider as our basic set of axioms a processor with a fixed instruction set, a set of registers, and a fixed amount of memory (store). The computer system that is going to use these axioms is shown in Fig. IV. 4. The system consists of three echelons, echelons zero and one each has one singleton. Echelon two has three singletons, all of them are slaves of the singleton in echelon one.

The hierarchy of state spaces (passive elements) has to be defined from the passive part of the axioms, namely:

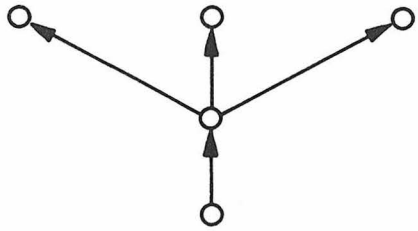


Fig. IV. 4. A Computer System.

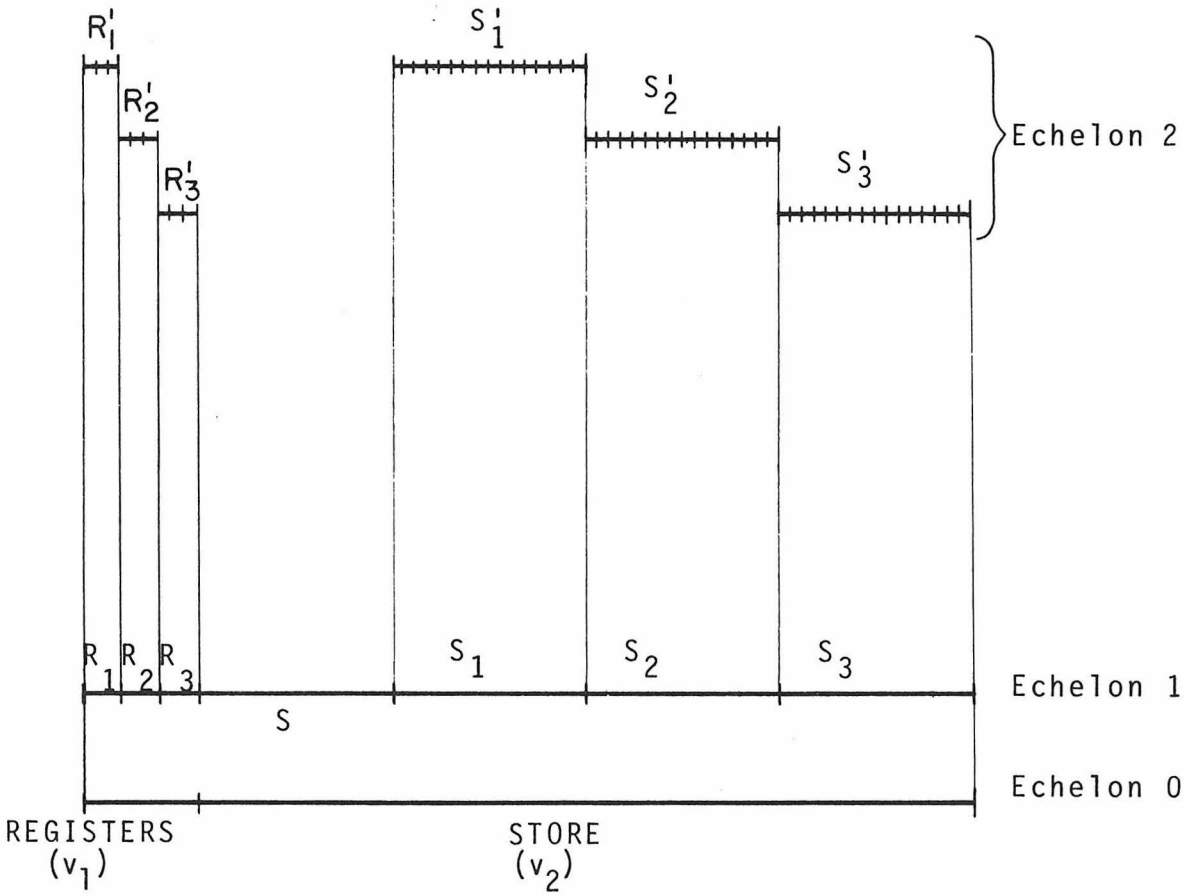


Fig. IV. 5. Hierarchy of State Spaces.

from the set of registers and the store. Remember that the basic idea in the definition of master space and slave space is that a single point in the space of the master is a set of points in the space of the slave. Fig. IV. 5 represents a hierarchy of state spaces for the system in Fig. IV. 4.

Fig. IV. 5 can be explained as follows: the state space of the singleton in echelon zero (singleton zero) is defined by two variables: v_1 = set of registers, and v_2 = store. However, the singleton zero does not operate on these variables, and as far as it is concerned these two variables only define one point in its space, which can be described as "anything in the registers and anything in the store." The singleton in echelon one (singleton one) moves within this single point of its master, but to it, the single point is a set of points defined by its internal variables which are (Fig. IV. 5):

S, R_1, R_2, R_3, S_1, S_2, S_3. However, the exact contents of R_1, R_2, R_3, S_1, S_2, S_3 are irrelevant to it because it does not operate on these variables, instead it only provides these variables to its slaves so that they can operate on the variables. Three points in the space of singleton one define the complete state space of its slaves, namely: (S_1, R_1), (S_2, R_2), (S_3, R_3), and within each slave each one of these points is considered as a complete state space, which can be described as a subset of the basic register set and a piece of the store (Fig. IV. 5).

We can easily see that the set of passive components of a computer system is a subset of the power set of the axioms. We can also see that low level echelons manipulate high level abstractions, so that in a sense echelon 0 can be considered highest. In other words, the lowest echelons can be considered to be the most abstract, as far as passive components are concerned.

The hierarchy of state transition functions is not the same as the one of state spaces. Fig. IV.6 illustrates this new hierarchy. The set I is the basic instruction set of the processor, and that is what exists at echelon 0. Echelon 1

echelon 2:	$A \cup f_1(A, B)$	singleton 1
	$A \cup f_2(A, B)$	singleton 2
	$A \cup f_3(A, B)$	singleton 3
echelon 1:	$I = A \cup B$	only singleton
echelon 0:	I	only singleton

Fig. IV. 6. Hierarchy of State Transition Functions.

partitions that set into subsets; the singletons at echelon 2 have direct access to only one of those subsets, and also by going into intermediate states (calling routines in echelon 1)

they have access to new instructions defined as a function of the instruction set of the previous echelon.

Then, the picture of the complete computer system is a combination of hierarchical data structures together with subsets and composition of functions.

V. A DESIGN EXAMPLE

The purpose of this chapter is to show how my formal structure can aid in the design and description of computer systems. The chapter is organized in two sections: the first one motivates the example by providing a user's overview; the second one is the analysis of the system itself. The system that I will present is called 'Experiment COntroller System' (ECOS), and it is a real time multi-processor computer system.

V. 1. User's Overview

The goal of ECOS is to provide a comfortable environment for the development of computer controlled experiments in psychophysics and neurophysiology. Typical experiments in these areas are: manipulation of text on a CRT screen as a function of the eye movements of a human subject, perception and learning tasks involving human and animal subjects, computation of characteristic functions of retinal electro-potentials, studies of brainwaves, measurements of reaction time, single neuron activity studies, etc.

The common characteristic of these experiments is that a stimulus is given to an organism, and the response of the organism is measured by acquiring data from sources that are considered meaningful; for instance, to study

retinal electro-potentials, lights of various intensities are flashed with different frequencies on a subject's eye, and an electrode on the surface of the eye is used to measure variations of electro-potentials on the retina. The stimulus is the light presented to the eye, and the response is the electro-potential evoked on the retina.

Then an experiment can be thought of as two related activities: stimulus presentation and data acquisition. These two activities are closely related because even in the simplest experiment one usually wants to acquire data only while the stimulus is being presented to the organism, and in more complex experiments, the required stimulus may be a function of the data being acquired, or the acquisition strategy may be dependent on the type of stimulus being presented. For instance, in the previous example about retinal potentials, a typical experimenter wants to measure the retinal potentials as they relate to the flashes of light, so the acquisition of data (retinal electro-potentials) is closely related to the stimulus (flashes); if the experimenter wanted to obtain a very specific response (like the determination of a threshold), then he would want to vary the stimulus until such a specific response is obtained.

A computer system to control experiments should provide simple tools to control the presentation of stimuli, and to control the acquisition of data. Fig. V.1 represents an experiment together with the computer system to

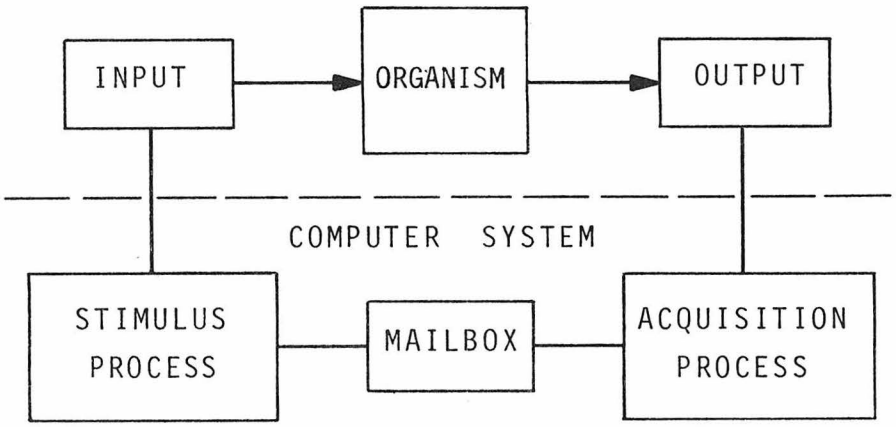


Fig. V. 1. ECOS: First User's View

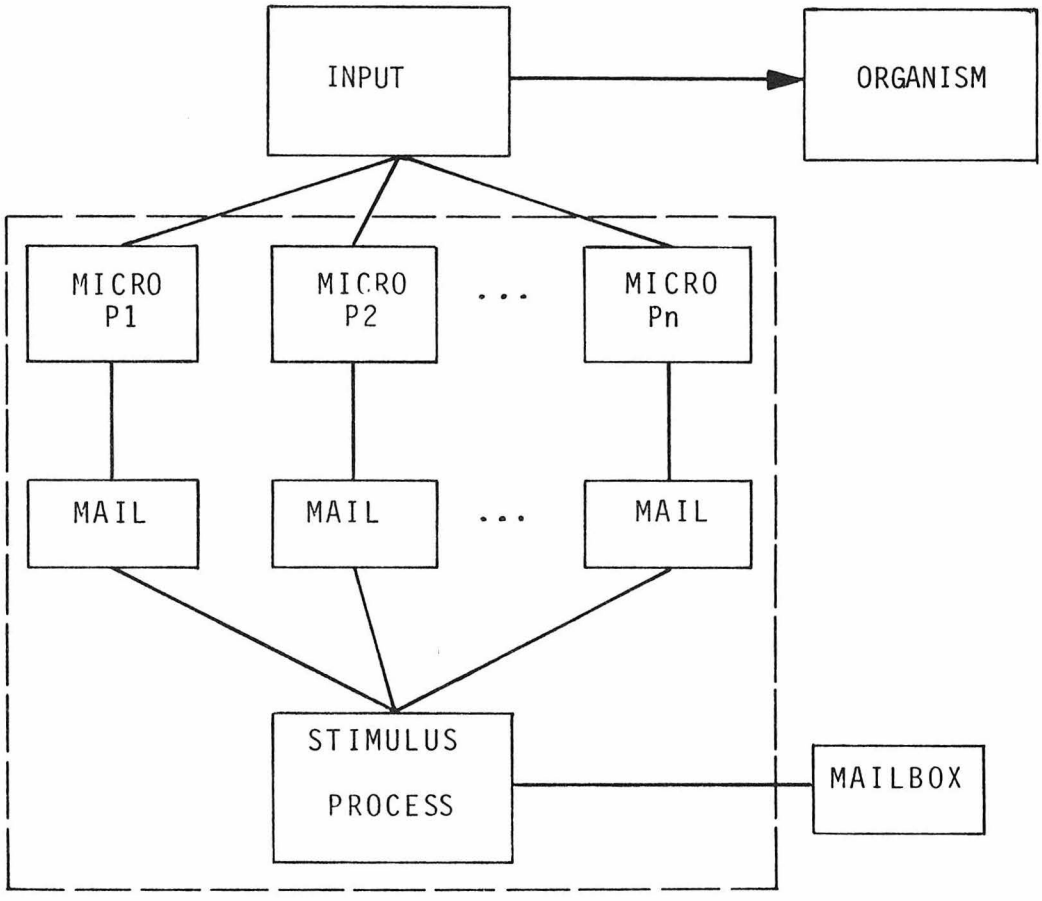


Fig. V. 2. ECOS: Second User's View

control it. The boxes labeled organism, input and output do not belong to the computer system; the boxes input and output represent the apparatus that can be controlled by the computer system to generate the actual stimulus (typical apparatus are: lamps, noise sources, motors, displays, etc.) and the actual acquisition of data (typical devices are: multiplexers, A/D's, filters, etc.). The arrows in Fig. V.1 indicate the direction of information to/from the organism; the solid lines (without arrowheads) represent paths of information that can be produced/consumed by the computer system. The stimulus and acquisition processes represent sequential processes within the computer system, and the box labeled mailbox represents a communication buffer between the sequential processes.

Notice that in general, the information path between the stimulus (or acquisition) process and the input (or output) box is multiple; the input (or output) box is composed of several apparatus, and so the stimulus (or acquisition) process has to manipulate several information channels.

If the experiment requires complex and dynamic timing relationships between channels, then a single sequential process to control several information channels is no longer appropriate, and in that case, the computer system should have a stimulus (or acquisition) group, as in Fig. V.2. Within such group, the function of the stimulus (or acquisition) process is to coordinate the

activities of the several micro-P's (micro-P1, micro-P2, ... micro-Pn), the function of each micro-P is to control a single information channel (or a very coherent group of information channels); because of these functional characteristics, in what follows I shall often refer to the stimulus (or acquisition) process as the stimulus (or acquisition) coordinator, and I shall often refer to the micro-P's as micro-controllers.

Between a coordinator and each micro-controller in Fig. V.2 there is a box called mail, its function is to provide a buffer for the exchange of information between a coordinator and a micro-controller.

Besides the interaction with the organism through the input and output boxes, the computer system also has to interact with the experimenter, because the experimenter has to set experimental parameters and may need to follow the experiment as it is going. He may also need to modify experimental conditions on the fly. Therefore a coordinator (Fig. V.2), i.e. stimulus or acquisition process, will manipulate information within the computer system, and it will also interact with part of the external environment, namely the experimenter.

Finally, the computer system should provide facilities for the manipulation and store of the data acquired during the experiment; typical services are filing services and i/o device drivers.

To summarize, some specific requirements of the computer system are listed in order of decreasing importance:

- Control of experiment's inputs and outputs should be very accurately timed.

- All timing and synchronization should be handled by the computer system; when the experimenter's requirements exceed the capabilities of the system, the experimenter should be warned.

- Interaction with the experimenter and use of the computer's peripheral devices should be as simple as possible.

- The number of stimulus and acquisition micro-controllers is not fixed, and will change from experiment to experiment.

- The services provided by the coordinators and the services provided by the micro-controllers should be as similar as possible. Services meaning: interaction between controllers and coordinators, interaction with the experimenter, and handling of computer's peripherals.

V. 2. Designer's View

In the last two chapters I mentioned that the function of the topmost singletons (leaves) is to interact with the external environment. The user's overview of ECOS (V.1) presents the different elements that are required to interact with the external environment (stimulus and acquisition supervisors and micro-controllers), therefore, there must

be a leaf corresponding to each of these elements, as Fig. V.3 indicates.

The user's requirements influence more than the leaves, first of all there are the communication requirements between the leaves; stimulus and acquisition coordinators should be able to communicate with each other, and each micro-controller should be able to communicate with either one of the coordinators. Fig. V. 4. illustrates these additional requirements; notice that the elements labeled 'mailboxes' are not leaves nor masters of the leaves, they are only services that the leaves can use, and as such they should be implemented by the master, or masters, of the leaves.

Before going any further, let me mention that if in Fig. V. 4 we write 'process' instead of leaf and 'mailbox monitor' instead of 'mailbox,' then Fig. V. 4 would represent a concurrent program[B3], that is, it would be a pure software view.

However, the design process should not stop with this concurrent program; there are additional user's requirements, namely: external speed of coordinators and controllers, and services provided.

The speed requirement is a very critical one; accurate timing is essential in most experiments. Therefore we have to explore how the master (or masters) of the leaves implement the slaves to use time (IV.2). Since a

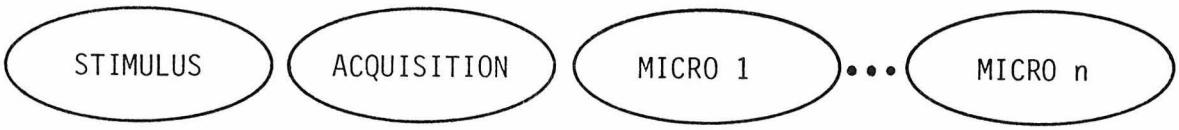


Fig. V. 3. System's Leaves

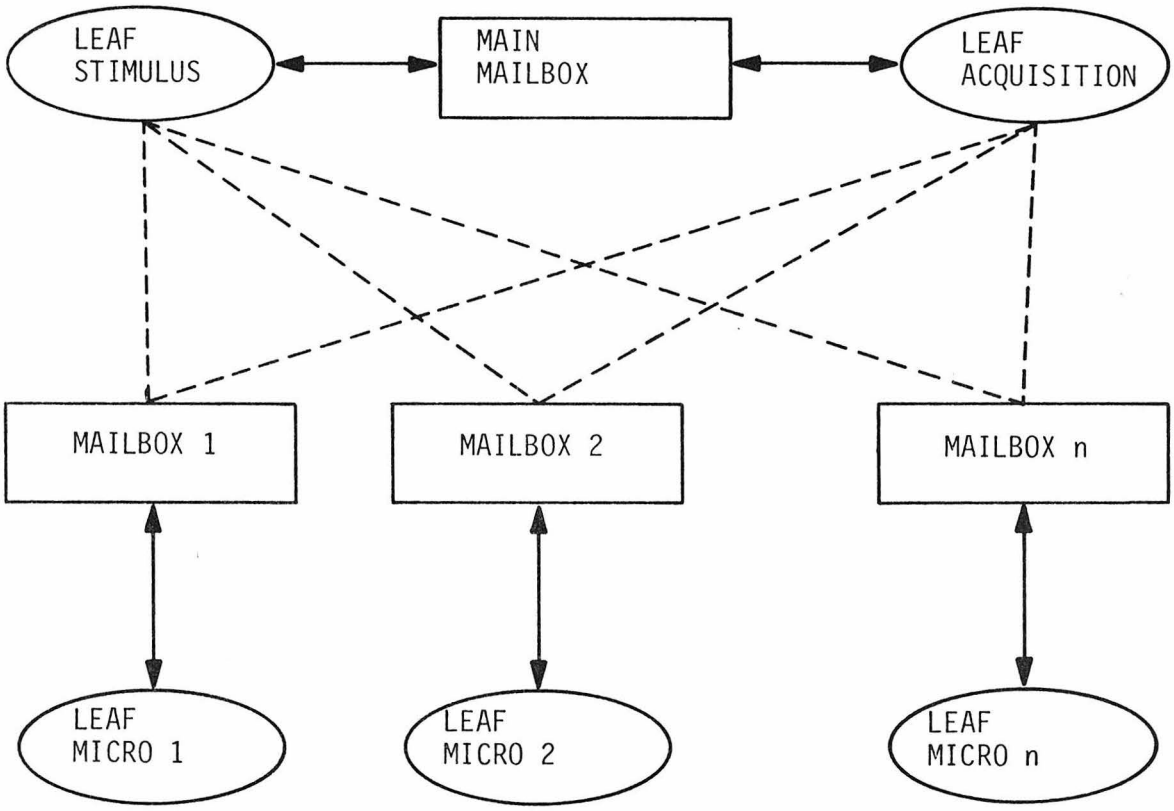


Fig. V. 4. ECOS Functional Structure
(= Software View).

single hardware processor can execute only one action at a time, a single processor can implement only concurrent slaves. It is very clear that a concurrent implementation cannot satisfy strict timing requirements (if it is required to execute two slaves at the same time, it cannot be done). Then a parallel implementation of slaves is required; that is, several hardware processors are needed.

From the user's view (V. 1), we know that the role played by the coordinators is different from the role played by the controllers. The controllers are the ones that have direct contact with the experiment; in other words, they are the ones that have strict timing requirements. On the other hand, the coordinators have no direct contact with the experiment, therefore timing is not critical, and hence they could be implemented as concurrent slaves. Furthermore, the controllers have a very small universe; each one of them is responsible only for a very specific and limited part of the experiment. Each coordinator has a wider view of the experiment; its task requires it to interact with several controllers and the other coordinator. Coordinators are also the singletons that interact with the experimenter, and they are the ones that accomplish more complex tasks.

Therefore, it is a reasonable design decision to use hardware processors of different power for the controllers and the coordinators, and also, to use a single, powerful processor for both coordinators. To put it simply, the

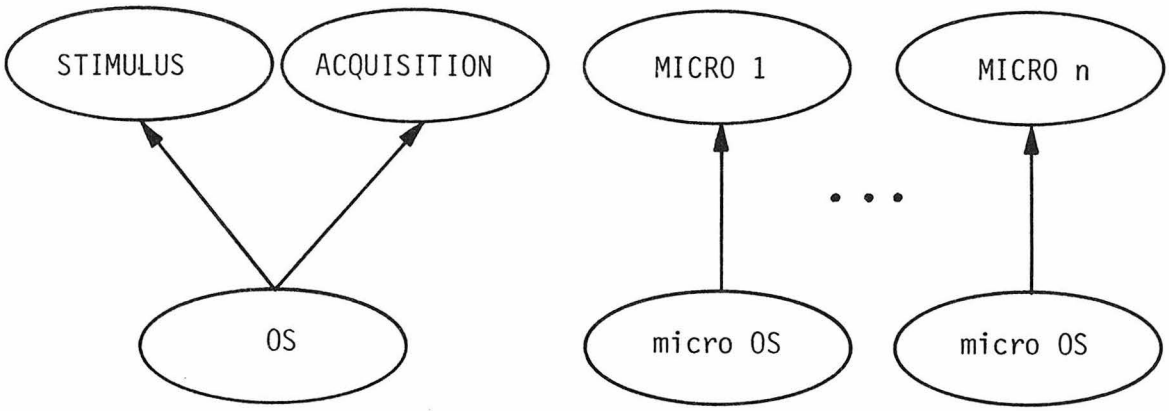


Fig. V. 5. Leaves and their Masters

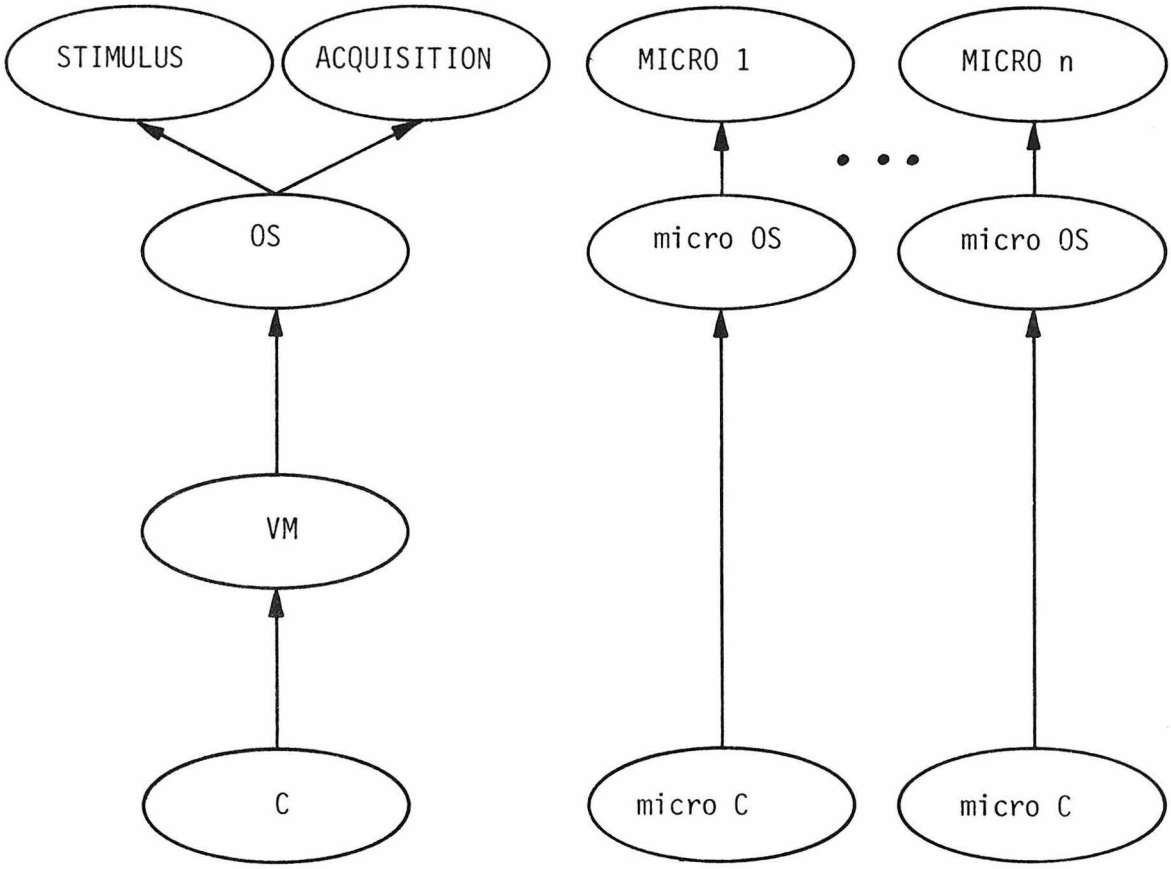


Fig. V. 6. From Hardware to Leaves

structure of Fig. V. 5.

Notice that the mailboxes of Fig. V. 4 are missing from Fig. V. 5; they have to be implemented by the singletons that are one level below the top. The masters of the leaves also have to implement the operations to manipulate computer's peripherals, and since those are the traditional roles of operating systems, I am labeling those singletons OS (operating system) and micro OS. Of course, these operating systems do not have to be implemented in software, but in the system that I am considering they are.

There is a very important difference between any micro OS and the OS, namely: the OS has two slaves and any micro OS only has one. The facilities required for the implementation of concurrent slaves could be directly provided by the OS; however, there is enough practical evidence[B2], that because of their nature they should be provided by a singleton of lower level than the OS*. This singleton, master of OS, is called a Virtual Machine (VM). Since the micro OS only has one slave, it does not require a virtual machine.

Bus oriented (hardware) computers provide an appro-

---

*Facilities to implement concurrent slaves are required to handle interrupts. A singleton of lower level than the OS can isolate the handling of interrupts from the rest of the computer system, thereby increasing the simplicity and reliability of the complete computer system.

priate environment for the VM and for the micro OS. Fig.V. 6 illustrates the system from the leaves down to the computers; the computer that provides the environment for the VM is labeled C, and it is more powerful than the one labeled micro-C which provides the environment for the micro-OS.

The system in Fig. V. 6 is not yet a computer system, since it is not connected. Furthermore, it does not satisfy essential requirements, namely: the controllers can not communicate with the coordinators, and the system's peripherals could be connected to only a single hardware computer. By adding one more singleton at the bottom as master of the hardware machines, the system can satisfy those last requirements (Fig. V. 7). The singleton L has one slave C and n slaves micro-C, all of these slaves run in parallel. The name L stands for link between all the different computers; L provides the services required by the computers for communication and sharing of resources.

Until this point the singletons have been only nodes of the system graph; it is time to start looking at the environment within which each singleton executes, and to study the internal structure of each singleton. None of the descriptions will be exhaustive, my purpose being only to illustrate the general approach; in this description I will proceed from echelon zero up to the leaves.

Echelon zero provides all the axioms on which the system is built, in this case those axioms are the hardware view of the system, as Fig. V. 8 illustrates. The computer

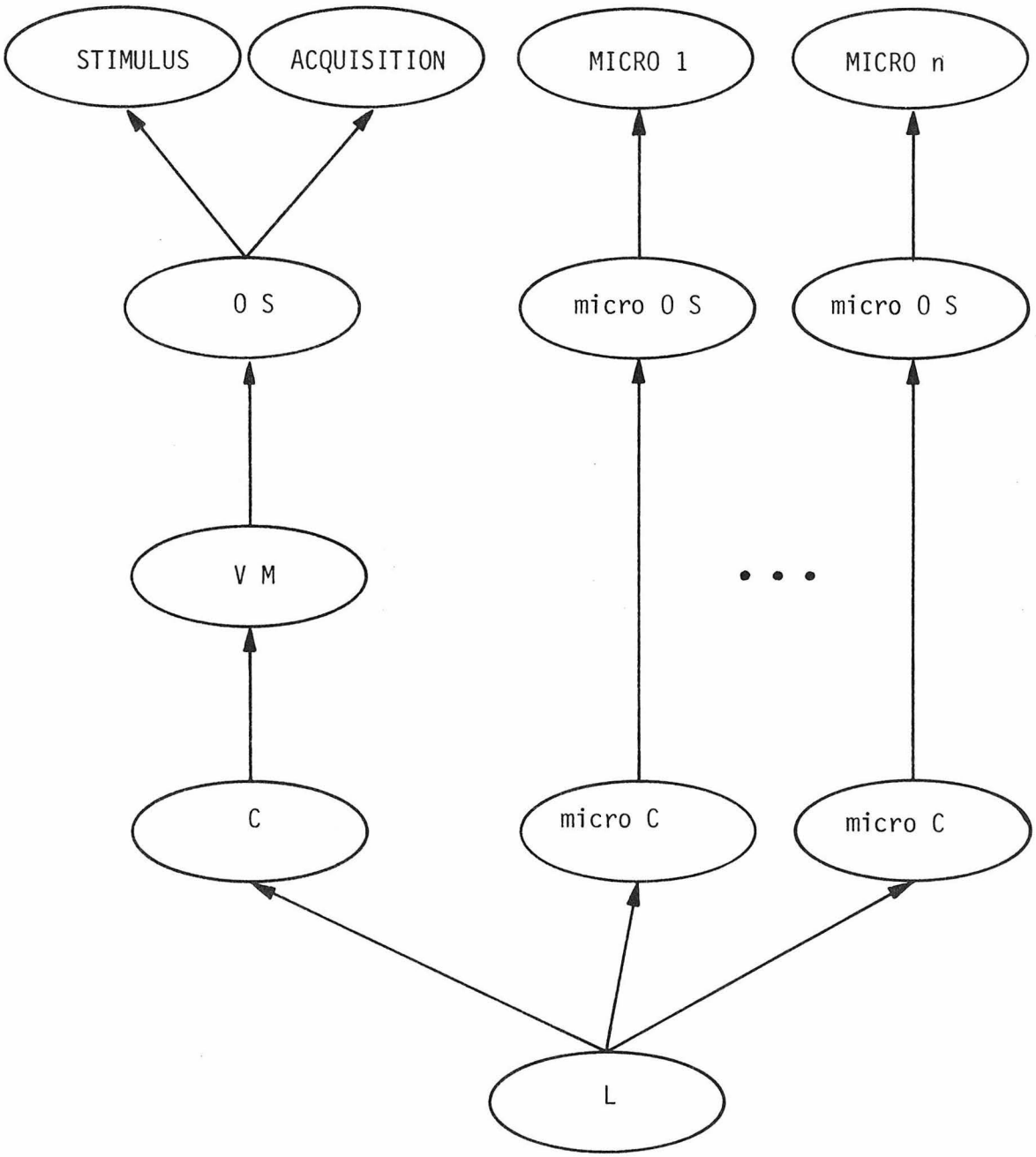


Fig. V. 7. ECOS Computer System

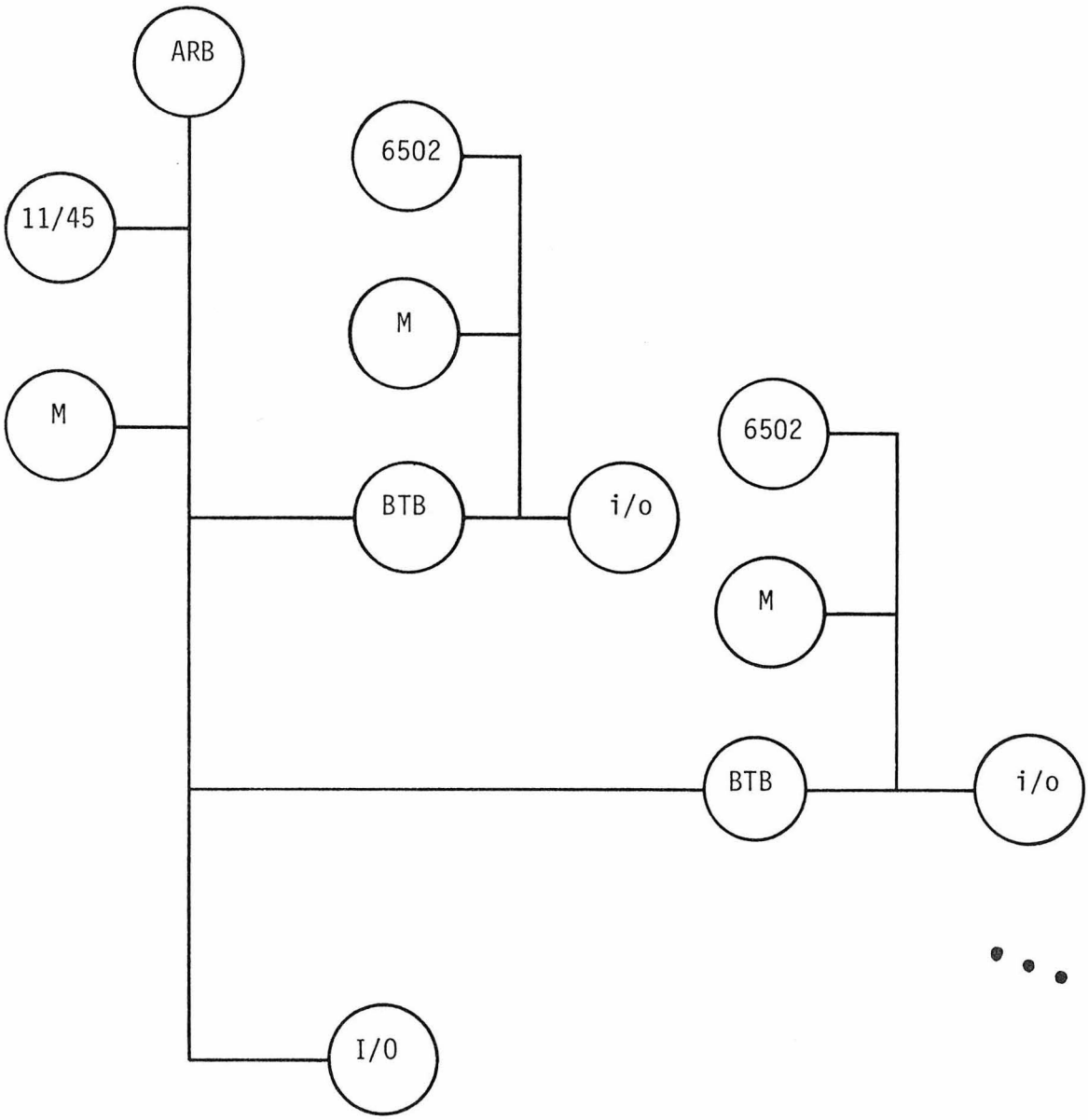
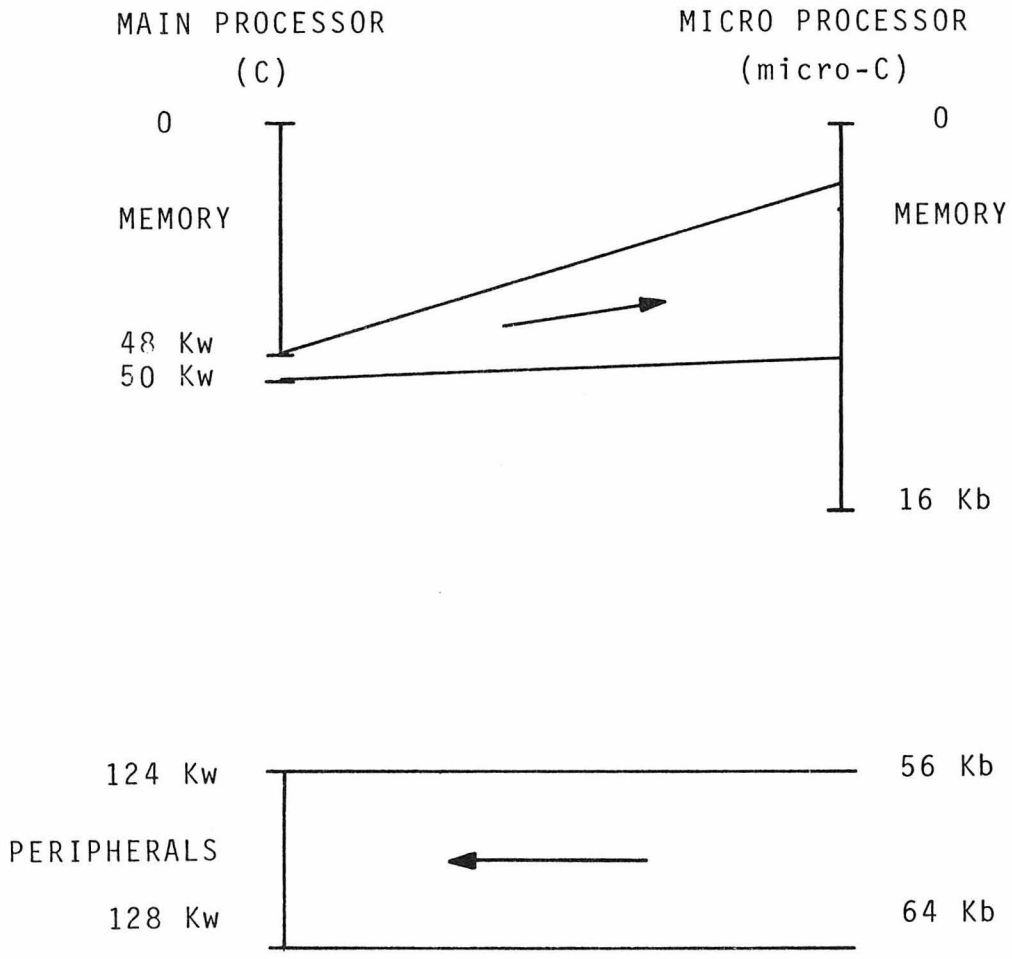


Fig. V. 8. ECOS Echelon Zero (= Hardware View)

C is a PDP 11/45[D5], the computer micro-C is an MCS6502[M4], ARB is the bus arbitration unit of the PDP 11, all M's are memories, i/o are private input/output devices, I/O are shared input/output devices, all BTB are bus-to-bus interfaces (similar to[F1] and "unibus window"[D6]),the solid lines are the different buses; different buses have a BTB in between.

Since the BTB interface is an essential mechanism in ECOS, I shall provide a short explanation of its function for the sake of completeness. The bus is a mechanism that provides an addressing space for a processor; C has an addressing space that goes from 0 to 128 Kw (kilo-words), anything which can be accessed by an address can be placed within such space. In particular, Fig. V. 9 shows that C (main processor) has memory placed in the range 0-48 Kw, and peripherals in the range 124-128 Kw, the range 50-124 Kw is not used, and finally the range from 48-50 Kw is used by the BTB. The function of the BTB interface is to translate addresses between different address spaces. Looking at Fig. V. 9, any read/write operation in the main processor space which falls within the range 48-50 Kw will be executed as a read/write operation in the micro-C space; likewise, for the range 56-64 Kb (kilo-bytes) in the micro-C space. The exact map from C to micro-C depends on a register of BTB, this register called map register can be set by micro-C, and therefore each micro-C defines the exact mapping of its BTB from C to itself. The map from micro-C to C is always as



(arrows indicate the direction of translation)

Fig. V. 9. Bus-To-Bus Interface

represented in Fig. V.9.

Physically, each BTB is composed of about 200 TTL integrated circuits, and its logic is partly synchronous and partly asynchronous because it joins an asynchronous bus (PDP 11) and a synchronous bus (MCS 6502).

Singleton L provides the environment for its consistent slaves to execute. The consistent slaves are one C and several micro-C's; the environment for each of them to move is defined by the buses, the BTB interfaces, the bus arbitration unit, and the shared peripherals. The environment of the slaves should not intersect for the slaves to be consistent; the BTB's and the arbitration unit on the PDP 11/45 bus prevent any intersection when using the buses. However, intersections when using shared peripherals have to be prevented by each shared device, i.e., each shared peripheral should queue the requests for its use. A major difficulty in the construction of ECOS is that commercial peripherals do not have any mechanism for queueing requests from different processors, therefore this problem has to be solved by a different singleton (by the OS). This brute force solution has proved itself to be a major flaw in the simplicity, reliability, and efficiency of ECOS. In an appropriate design, the state space of L is defined by the BTB's map registers and by the shared peripheral's owners; the state transition function is composed of the operations that set BTB's map registers and by operations that request peripheral ownership. The second group of operations is missing from my

implementation of ECOS.

The structure of the singletons of echelon 1 (C and micro-C) is given elsewhere[D5, M4]; for the purpose of this thesis it is enough to say that each singleton of echelon 1 builds on top of echelon zero a new state space defined by the so-called processor status, register set and memory; and it also builds a new state transition function called the instruction set. Since these hardware definitions can be considered as new axioms, it could be said that the axioms of echelon zero are extended or refined by new axioms at echelon 1; however, my point of view is that echelon zero provides the specific requirements and facilities for echelon 1, and echelon 1 is just a specific design built on top of echelon zero.

With respect to the descriptions provided for the commercial computers that constitute echelon 1, I should point out that they are very incomplete, and in some cases even erroneous. Even the best mathematical formalism will be useless until hardware and software systems are carefully and honestly specified and documented.

The singleton C provides the environment for the Virtual Machine (VM) to run. The main function of the VM is to provide the facilities needed for the implementation of concurrent slaves, namely: handling of interrupts and processor multiplexing. Internally the VM is composed of two elements: the kernel and the interpreter. The kernel

is in charge of handling interrupts, multiplexing C, and providing basic drivers for the peripheral devices; the interpreter uses the instruction set provided by C to define a new instruction set (121 operations) which is more appropriate for the higher singletons.

The state space of the virtual machine is defined by the variables of its components; one variable is particularly important, and it is the one that keeps track of which slave is running at a particular time. The low level representation of the VM is PDP 11/45 machine code, the high level representation of the VM is written in a combination of PDP 11 macro-assembler and Pascal-like comments. The size of the low level representation is about 3 Kw (kilowords).

The VM has only one slave which is called OS. The low level representation of OS is virtual code, i.e. code which can be manipulated by the part of the VM called the interpreter; the high level representation of the OS is written in Concurrent Pascal[B3]; to go from the high level to the low level representation, a compilation process is required. The size of the high level representation is about 3000 lines, the size of the low level representation is about 15 Kw (virtual code).

The internal structure of OS is represented in Fig. V.10; the boxes labeled 'proc' represent processes, and the boxes labeled 'mon' represent monitors. OS has two pro-

cesses, the process stimuler* provides the environment for the stimulus coordinator, and the process acquirer provides the environment for the acquisition coordinator. Both processes have access to 5 monitors; the system manager keeps track of which process or processor is using each shared peripheral, i.e. this monitor implements some of the services that should be provided by echelon zero. The monitors 'system tty' and 'system disk' prevent simultaneous access to those devices. The monitor 'main mailbox' implements the mailbox between the stimulus and acquisition coordinators. Finally, the monitor 'system watches' provides a set of watches with a common timebase, namely the system real time clock.

The internal structure of the processes stimuler (Fig. V.10) and acquirer is the same, it is called a server. The internal structure of the server is represented in Fig. V.11, it is a hierarchy of classes which implements all the mechanisms needed to manipulate files, interact with the experimenter (classes 'hard copy files' and 'magnetic files'), interact with the micro-OS (class microprocessors), and load and execute sequential programs (class loader).

Table V.1 lists the different components of OS, its high level representation size, and the number of bugs found during testing. The different components were tested in the

---

*I have use the word 'stimuler' instead of 'stimulator' to emphasize that stimuler is not an apparatus.

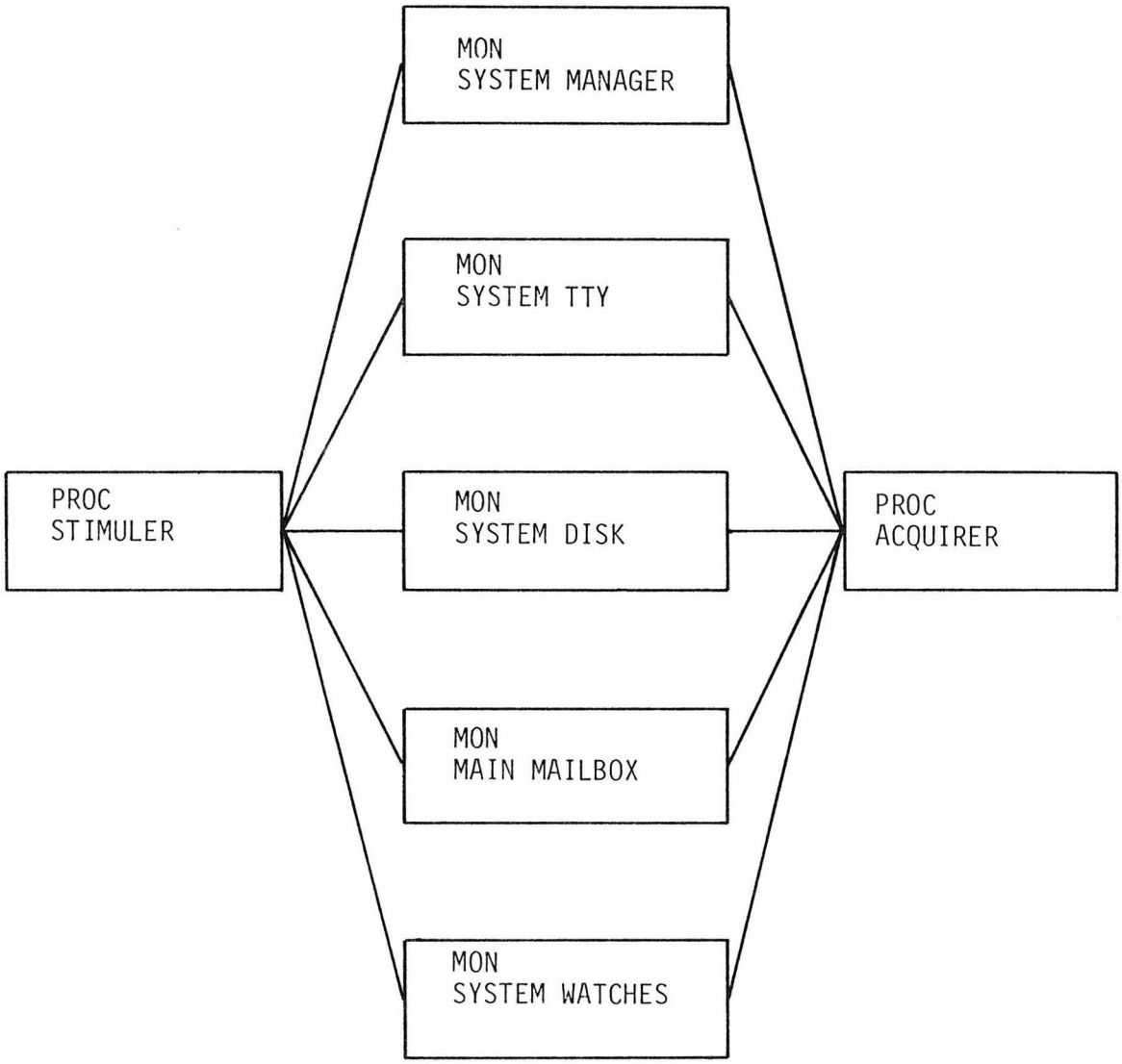


Fig. V.10. OS Internal Structure

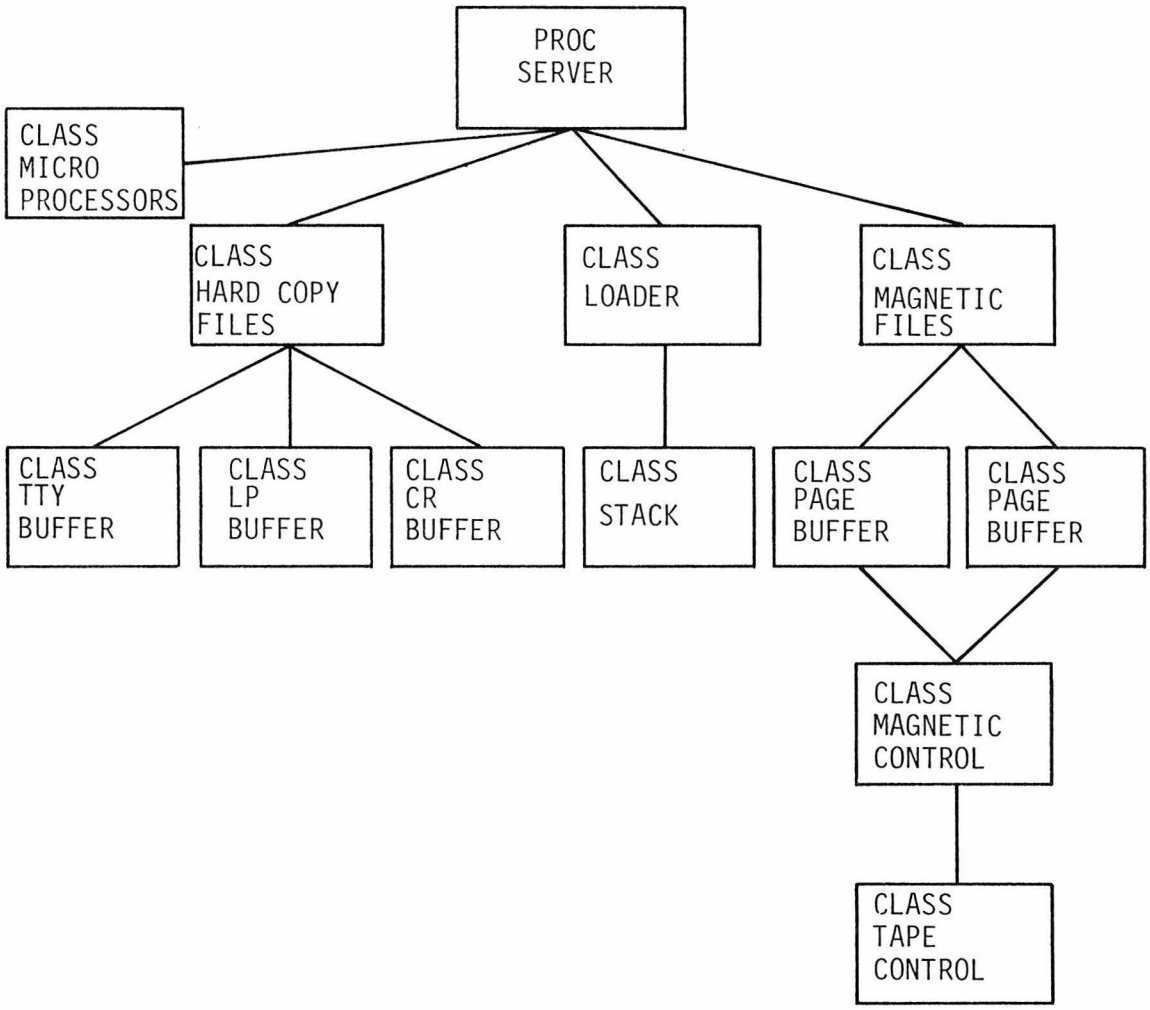


Fig. V.11. OS Server Process Structure

COMPONENT	SIZE (# lines)	BUGS FOUND
system tty	74	0
tty buffer	117	1
lp buffer	52	0
or buffer	60	0
hard copy files	445	6
system disk	145	1
tape controller	111	0
magnetic controller	110	0
page buffer	215	3
magnetic files	220	3
main mailbox	264	2
system manager	96	1
system watches	49	0
stack	63	1
loader	99	0
server	513	0
microprocessors	116	1

a total of 19 bugs in 3000 lines of high level representation

i.e. 1 bug/158 lines or 1 bug/790 words of virtual code

TABLE V.1. OS Components Statistics

order given in the table (top row first, bottom last). Notice that for debugging a hierarchical singleton we start from the leaves of its internal hierarchy; however, for debugging a computer system, we start with the singleton that is absolute master of the hierarchical computer system. (Do not confuse internal structure of a singleton with the structure of the computer system nor with the hierarchy of representations of a singleton.)

The state space of OS is defined by the variables of its components. OS has two slaves, and it provides two equal environments which are defined by the prefix that is summarized in Fig. V.12. The prefix defines some implementation parameters of the high level language, types and constants which are specific to OS, and services implemented by OS; the services shown in Fig. V.12 are basic input/output (from 'readi' to 'plot'), file management (from 'reserve' to 'endoffile'), synchronization and message passing between servers (from 'synchro' to 'waitandget'), timing (from 'resetwatch' to 'watchelapsed'), and interaction with micro-controllers.

A stimulus or acquisition coordinator has a transition function defined by most of the 121 instructions provided by VM, and a total of 48 services provided by OS; most of the instructions provided by VM produce state transitions only in the state space of the stimulus or acquisition coordinators, however, some instructions also

```
    " IMPLEMENTATION PARAMETERS "
CONST SETLENGTH = 127;
TYPE ANYSET = SET OF 0..SETLENGTH;
CONST LINELENGTH = 132;
. . .
    " FILES "
TYPE
    IOFILE = (PRINTERFILE, DISKFILE0, DISKFILE1, TAPEFILE0,
              TAPEFILE1, TTYFILE, CARDSFILE);
    INFILE = DISKFILE0..CARDSFILE;
    OUTFILE = PRINTERFILE..TTYFILE;
    " SERVICES "
PROCEDURE READI (SOURCE: INFILE; VAR I: INTEGER);
PROCEDURE READC (SOURCE: INFILE; VAR C: CHAR);
FUNCTION LASTC (SOURCE: HARDINFILE): CHAR;
. . .
PROCEDURE WRITEI (DEST: OUTFILE; I: INTEGER);
PROCEDURE WRITEC (DEST: OUTFILE; C: CHAR);
. . .
PROCEDURE PLOT (CURVE: CURVETYPE; POINTS: CURVEPOINTS);
. . .
PROCEDURE RESERVE (IOUNIT: IOFILE);
PROCEDURE RELEASE (IOUNIT: IOFILE);
FUNCTION ENDOFFILE (SOURCE: INFILE): BOOLEAN;
. . .
PROCEDURE SYNCHRO;
PROCEDURE SETFLAG (FLAGLABEL: LABELUNIT);
FUNCTION FLAG (FLAGLABEL: LABELUNIT): BOOLEAN;
. . .
PROCEDURE SEND (HEAD: MESSAGEHEAD; M: UNIV INTEGER);
PROCEDURE SENDANDWAIT (HEAD: MESSAGEHEAD; M: UNIV INTEGER);
PROCEDURE GET (HEAD: MESSAGEHEAD; VAR M: UNIV INTEGER; VAR...
PROCEDURE WAITANDGET (HEAD: MESSAGEHEAD; VAR M: UNIV INTEGER);
. . .
PROCEDURE RESETWATCH (WATCHNO: WATCHUNIT);
PROCEDURE LOOKWATCH (WATCHNO: WATCHUNIT; VAR COUNT: INTEGER);
FUNCTION WATCHELAPSED (WATCHNO: WATCHUNIT; COUNT: INTEGER):...
. . .
PROCEDURE RESERVEMICRO (UNITNO: MICROUNIT);
PROCEDURE RELEASEMICRO (UNITNO: MICROUNIT);
PROCEDURE LOADMICRO (UNITNO: MICROUNIT; NAME: IDENTIFIER);
PROCEDURE SENDMICRO (UNITNO: MICROUNIT; DATA: UNIV INTEGER);
PROCEDURE SENDANDWAITMICRO (UNITNO: MICROUNIT; DATA: UNIV...
PROCEDURE GETMICRO (UNITNO: MICROUNIT; VAR DATA: UNIV INT...
. . .
```

Fig V.12. Stimulus or Acquisition Environment (Sequential Pascal Syntax)

require state transitions in OS and VM. All of the services provided by OS require state transitions in the state space of OS, and some also require state transitions in the state space of VM. A stimulus or acquisition coordinator has a large velocity if it rarely requires changes in the state of OS or in the state of VM. However, if a stimulus or acquisition coordinator often requires services provided by OS, then its velocity will be small.

A stimulus or acquisition coordinator cannot manipulate magnetic disc sectors, tracks and surfaces; instead, it can only read or write disc pages; a similar situation occurs with all others peripherals. The total number of states that a stimulus or acquisition coordinator can reach has been reduced by the hierarchy of VM and OS; the transparency of a stimulus or acquisition coordinator is small.

Continuing with the analysis of the ECOS computer system, we have the only slave of the micro-C, called micro-OS. The purpose of the micro-OS is to provide an environment for a micro-controller. The internal structure of the micro-OS is a very simple hierarchy of classes that provide basic input/output tools, and message passing services between the micro-controller and the coordinators.

The low level representation language of micro-OS is MCS6502 machine code, its high level representation language is micro-assembler together with comments in a Pascal-like notation. The size of the micro-OS low level

representation is about 1 Kb, and the environment that it provides for a micro-controller is shown in Fig. V.13 (compare with Fig. V.12).

Finally, the internal structure of the leaves of the computer system can not be given, because the experimenter that uses ECOS is the one that specifies the structure of the leaves.

```
;   USER PROGRAM ALLOCATION

    .DEF USERVARBASE := 0
    .DEF USERCODEBASE := 2000
     . . .
;    TEMPORARY BUFFERS

    .DEF PCLBUFFER := 0EF
    .DEF PCHBUFFER := 0F0
    .DEF PBUFFER:= 0F1
     . . .
;   SYSTEM PARAMETERS

;   IO PORT 1 PARAMETERS

;   .DEF PORT1ADATA := 1700
    .DEF PORT1ADIRECTION := 1701          ; 0-IN, 1-OUT
     . . .
;    TIMER PARAMETERS

    .DEF TIMEDIV1    := 170C    ; 1 US TO 256 US
    .DEF TIMEDIV8    := 170D    ; 8 US TO 2.048 MS
     . . .
;   PROCEDURES ENTRY
;   CALL BY USING BRK          ; PROCEDURE HALT;
    .DEF GOTIMER := 22E        ; PROCEDURE GOTIMER;
    .DEF LOOKTIMER := 23D      ; PROCEDURE LOOKTIMER (VAR A:...
    .DEF AWAITTIMER := 241     ; PROCEDURE AWAITTIMER;
    .DEF SEND := 2A9           ; PROCEDURE SEND (A: ACCUMULATOR);
    .DEF SENDANDWAIT := 2E0    ; PROCEDURE SENDANDWAIT (A: ...
    .DEF ANYMESSAGE := 264     ; FUNCTION ANYMESSAGE: BOOLEAN;
    .DEF GET := 2EB            ; PROCEDURE GET (VAR A: ...
    .DEF WAITANDGET := 318     ; PROCEDURE WAITANDGET (VAR A:...
    .DEF EMPTYMAIL := 321      ; PROCEDURE EMPTYMAIL;
    .DEF INITMICROOS := 3F4    ; PROCEDURE INITMICROOS;
    .DEF LOADNEXT := 480       ; PROCEDURE LOADNEXT;
```

Fig. V.13. Micro-controller Environment
(Micro-assembler Syntax)

VI. DISCUSSION

In the introduction, I stated that the purpose of this thesis was "to develop a formal structure which can be used to describe and to design general computer systems based on the sound principle of hierarchical structuring." I believe that such a purpose has been accomplished. Even though I have not given a formal proof that my structure is capable of representing any computer system, the concepts of state space, transition function and program (i.e., a singleton) are powerful enough to represent any automata; furthermore, the example of Chapter V demonstrates that a connected set of singletons is a very powerful and convenient tool because it abstracts all the essential features of a computer system and, therefore, it simplifies enormously the tasks of analysis and design of computer systems.

My structure provides, for the first time, a comprehensive treatment of hardware and software as a coherent unit. My approach uses some standard tools from systems theory, however I have developed some novel tools of my own, namely: a hierarchy of state spaces, transition functions, and time bases. In particular, the simplified treatment of time eliminates from the designer's mind the problem of mutual exclusion.

Mine is the first formal, engineering-oriented

definition of computer systems; my formal approach provides better conceptual tools than previously available for the analysis and design of computer systems; furthermore, within my formal framework the process of design and compilation can be easily explained and characterized. Formal measures of the complexity and velocity of a hierarchical system have been developed, providing for the first time precise tools that can be used to compare design alternatives.

However, my main contribution can be summarized as conceptual simplicity. Let me quote Simon:

> How complex or simple a structure is depends upon the way in which we describe it. Most of the complex structures found in the world are extremely redundant, and we can use this redundancy to simplify their description. But to use it, to achieve the simplification, we must find the right representation.

Computer structures are very redundant; my formal concepts take advantage of such redundancy and so they simplify the description of a computer. I do not know if it is even possible to find "the right representation" for computer systems; different contexts may not be totally satisfied by the same representation. But, in the context of overall systems' description and design, my representation is probably in the right direction because a singleton summarizes the overall important features of any computer element, and, furthermore, a connected set of singletons

clearly expresses the relationships between the different elements of a computer system.

The power of abstraction lies on the elimination of unessential details; however, essential details should be clearly expressed by the abstraction. A detail which is considered essential in computer systems design is the external world velocity of the system. My formal structure separates internal and external velocity, and within this thesis only internal velocity is formalized. External velocity is barely used in the example of Chapter V because the velocity requirements of different experiments are so different that only the most general solution is acceptable (one hardware processor per task). The fact that external velocity is not carefully studied at this time does not mean that it has been eliminated from the model; one of my future goals is to develop system velocity equations that relate internal velocities (which can be associated with the branches of the system graph) to external velocities (which are related to the absolute masters and leaves of the system). These velocity equations could be used to optimize the internal configuration of the system so that a maximal external velocity is achieved.

There is an additional benefit from any kind of formal treatment: a solid reference framework. Whether my structure is or is not the best possible for its intended

task, it does provide a solid context for the study of computer systems. In fact, one of the main motivations behind this thesis was to develop such a framework in order to be able to deal with basic design problems.

Let me elaborate on the kind of problems that I am interested in solving, and how they fit within my formal framework:

What is the optimal configuration of a computer system? In other words, is it possible to develop and use system velocity equations and transparency equations to maximize external and internal velocities, and at the same time minimize transparency so that the system is the fastest and simplest possible? Since a computer system is a graph, velocities and transparencies can be associated with its branches, and then combined at its nodes to satisfy conditions from the root(s) and leaves.

Now, assuming that optimal configurations can be developed, how do they affect technology, cost, reliability, and maintainability? i.e., assuming that a computer structure satisfies the basic user's requirements, maximizes velocity and minimizes transparency , can such a system be built? What kind of components are needed to build it? Is it also possible to minimize cost, and maximize reliability and maintainability? Specifically, what kind of singletons are needed and what kind of services should be offered by their

masters? (instruction sets, memory configurations, inter-processor interfaces, data structures, compilers, statements, ...); assuming that the optimal system can be built, can we also optimize configuration-cost, configuration-reliability, configuration-maintainability? that is, can we develop singletons as software and hardware 'computer modules,' assign cost, reliability and maintainability values to each module, and then from those values and the system graph optimize system cost, reliability, and maintainability?

I believe that the answers to those previous questions are all affirmative. In fact, if we forget about optimizing requirements, the system described in Chapter V (ECOS) is a small step in the right direction, since each echelon provides a set of modules that can be used for the design of different kinds of computer systems. For instance, if we consider the VM (Virtual Machine) and two micro-OS (micro-Operating System) they can be easily used to implement a spooling system in which one micro-OS takes care of all the input, the VM takes care of the compilation, and the other micro-OS takes care of the output.

Notice that once we have the computer modules and optimizing equations, the design process is straightforward, so simple indeed that an automatic system designer begins to be a real possibility.

I believe that the problems just discussed will have great influence in computer engineering, and my formal structure will prove itself to be a valuable tool in computer science.

APPENDIX A.

ILLUSTRATION OF BASIC CONCEPTS

As an example of a system, let us consider a three bit counter (Fig. A. 1); each one of the boxes (b_i) can

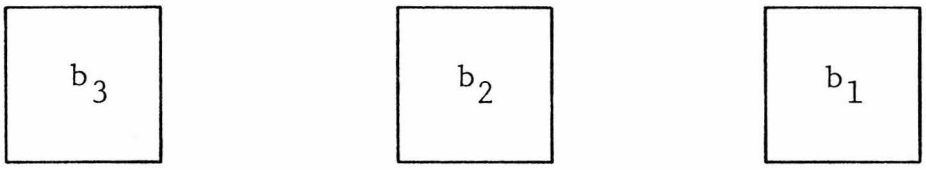


Fig. A.1. Three Bit Counter.

contain a 0 or a 1. Then the counter (three boxes) can be in any of the conditions shown in Table A. 1. where the three consecutive digits represent the contents of the three boxes; for instance, 100 means that box b_3 contains 1, and boxes b_2 and b_1 contain 0.

000	100
001	101
010	110
011	111

Table A. 1. State Space

The set of conditions represented in Table A. 1 is called the state space of the counter. Each condition is called a state of the counter, and it is considered a point in the counter's state space.

Let us add one more box, which we will call the program box; the contents of this box can also be a 1 or a 0, but it is not a part of the counter. Let us consider the program box as a time function, that is, its contents are a sequence of 1's and 0's, which change as a function of time. Time is a set that gives order to the changes. Here, the time set will be the natural numbers (0, 1, 2, ...).

How does the time function program affect the state of the counter? To answer this question we need to know the state transition function of the counter. The state transition function is a function of the state and the program, and maps into another state; this function is given in Table A. 2.

state	program →	state	state	program →	state
000	1	001	000	0	000
001	1	010	001	0	001
010	1	011	010	0	010
011	1	100	011	0	011
100	1	101	100	0	100
101	1	110	101	0	101
110	1	111	110	0	110
111	1	000	111	0	111

Table A. 2. State Transition Function.

The meaning of the state transition function is very simple: if the program is 1, then count; it it is 0

then stay the same.

Let us examine the behavior of the system as a function of time. When time is zero (nothing has happened), the counter has to have an initial state, say 011. Consider a time function program equal to 0,1,1,1,0,0,1,0,1,1,0 ; then Table A.3 can be easily obtained by using the program

time	state	program
0	011	0
1	011	1
2	100	1
3	101	1
4	110	0
5	110	0
6	110	1
7	111	0
8	111	1
9	000	1
10	001	0
11	001	

Table A. 3. Space Trajectory.

and the state transition function. The sequence of states that is generated by the program is called a trajectory in the state space. Notice that the state at time 11 is known even though the program is not given for that time.

The trajectory in Table A. 3 is a very particular one; it depends on a specific program and on a specific initial state. All sequences of 1's and 0's form a class of programs for this counter. The state space and the state transition function form an environment for a class of programs. The environment, together with a particular

program, form a singleton.

Let us now add one more box to our basic counter; so that it has four bits. Let us define a new state space for this new system as shown in Table A. 4.

1XXX	0XXX

(where X means "don't care")

Table A. 4. New State Space.

In this new space, the states 1111 and 1000 are equal. Then notice that a single point in this new space contains all the points of the old space for the three bit counter (Table A. 1.). This situation is represented in Fig. A. 2.

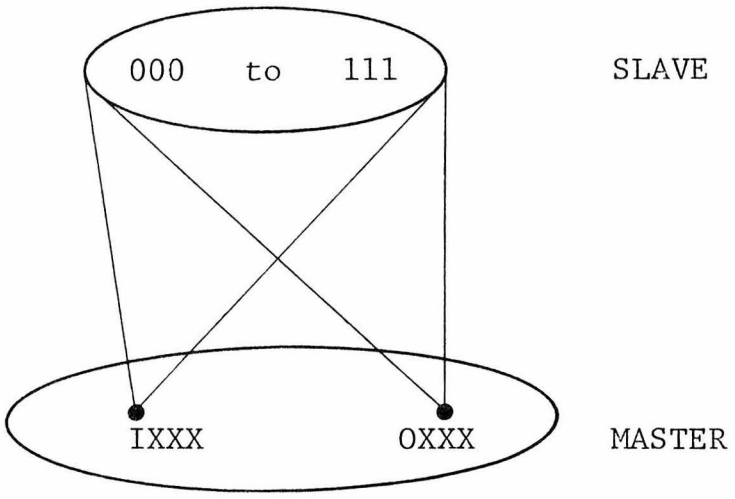


Fig. A. 2. Relation between Spaces.

Furthermore, let us modify the state transition function of Table A. 2 only when the state is 111, as shown in Table A. 5.

state	4th. bit	program	→	state	4th. bit
111	0	0		111	1
111	0	1		000	0
111	1	0		000	0
111	1	1		111	1

Table A. 5. New Transition Function.

This means that, in order to compute the transition when in state 111, we need to 'look' at the 4th. bit, and to use Table A. 5 instead of Table A. 2. The state 111 is an intermediate state; to compute the transition from an intermediate state we have to go from the slave space to the master space and back.

When each of the two systems operates in its own space, they move in different time scales; one may go through several states while the other does not move. Time only changes when the state changes through its transition function (sometimes a state may change to itself).

As a final observation, notice that the 'boxes' from Fig. A. 1 could contain data structures much more complex than one bit; that is, they could contain sub-spaces for other slaves.

APPENDIX B. ON CONCURRENT PROGRAMMING

This Appendix is not intended to be an exhaustive study of concurrent programming; its intention is only to illustrate what the problem is, and how to deal with it. A more complete discussion can be found in[B7].

A sequential program (Fig. B. 1.) is a set of statements and data such that only one statement at a time operates on the data.

DATA	D1	STATEMENTS	S1
	D2		S2
	...		...
	Dn		Sn

Fig. B. 1. Sequential Program

A concurrent program (Fig. B. 2.) is a set of two or more sets of statements and one set of data such that any number of statements at a time operate on the data; the only restriction is that within each set of statements only one statement at a time operates on the data, i.e., if there are n sets of statements, then at any given time there

STATEMENTS		DATA		STATEMENTS	
A	$S1^A$		D1	B	$S1^B$
	$S2^A$		D2		$S2^B$
	...		...		...
	Sm^A		Dn		Sp^B

Fig. B. 2. Concurrent Program

can be a maximum of n statements and a minimum of 1 statement operating on the data.

In this Appendix each set of statements in a concurrent program will be called a process* . There are no restrictions regarding the relative speed of the different processes.

Notice that a concurrent program is not a set of sequential programs; the main characteristic of a concurrent program is the possibility of having several operations on the same data at the same time. This possibility,named the mutual exclusion problem,is the main issue in concurrent programming.

A classic example of the mutual exclusion problem is a producer-consumer concurrent program which can be stated as follows: consider a concurrent program with two processes, one called producer, the other called consumer; both processes operate on an array of numbers which is called a buffer. The producer writes on the buffer starting with the first number and proceeding number by

---

*In general, a process is not only a set of statements, but also contains private data, that is, data which cannot be used by any other process. Since my purpose at this time is to discuss the problems that occur when accessing shared data, I will not consider private data.

number until it reaches the last, at that time it goes back and starts writing the first number again. The consumer reads from the buffer following the same pattern as the producer, i.e., it reads from first to last and then back to the first... Let me emphasize that there are no restrictions on the processes' speed.

If the producer is faster than the consumer, then parts of the buffer are going to be written over, and the consumer will get erroneous data. If the producer is slower than the consumer then parts of the buffer are going to be read over, and again, the consumer will get erroneous data. In general, any constant or variable difference in speeds will yield erroneous data.

Brute force concurrent programming is a 'technique' in which one only hopes that there won't be any difference in speeds. Obviously, this is not a very successful technique.

One good solution to this problem is to associate a special synchronization variable with the buffer. This special variable is called a semaphore, and it is such that only one process at a time can operate on it. There are two operations defined in a semaphore: signal and wait. When a process A does a wait operation on a semaphore, its execution will be delayed until another process B does a signal operation on the same semaphore; if the process B

has done the signal before process A does a wait, then process A will continue immediately.

In the producer-consumer example, the buffer requires two semaphores; one semaphore (call it producer_go) will tell the producer when the buffer has been read, the other semaphore (call it consumer_go) will tell the consumer when the buffer has been written. Fig. B. 3 shows this solution; write_buffer and read_buffer are procedures which operate on the shared variable buffer.

```
producer:                    consumer
  loop                         signal(producer_go)
    wait(producer_go)          loop
      write_buffer               wait(consumer_go)
    signal(consumer_go)            read_buffer
  end_loop                     signal(producer-go)
                               end_loop
```

Fig. B. 3. Semaphore Solution.

A different solution is called a monitor. A monitor is a data structure together with the definition of the operations that can be executed on the data structure (Fig. B. 4), and a very important restriction: only one process at a time can operate on the monitor. Each operation on the data structure is usually composed of

```
MONITOR
    data structure
    operations
END_MONITOR
```

Fig. B. 4. A Monitor

several statements; therefore, a process can execute several statements on a monitor data structure without any other process using the same data structure at the same time. A monitor can be thought of as a set of operations and a semaphore that tells if there is any process using any operation; however, the signal and wait operations on this semaphore are automatically handled by the monitor itself.

The problem of producer-consumer can be easily solved by declaring a buffer monitor with two operations: write and read. To write on the buffer the producer only says buffer.write , and to read from the monitor the consumer only says buffer.read . All the synchronization and mutual exclusion is handled by the monitor. Notice that the programmer still has to define the synchronization of the operations read and write, i.e., when reading wait until there are new data, when writing wait until the previous data have been read. The advantages of the monitor are: the mutual exclusion is automatically handled by the monitor itself, and the details of the operations are clearly separated from the use of the operations.

LIST OF REFERENCES

B1. Bell, C.G., Newell, A., "Computer Structures: Readings and Examples," McGraw-Hill, 1971.

B2. Brinch-Hansen, P., "Operating System Principles," Prentice-Hall, 1973.

B3. _____. "Concurrent Pascal Introduction," Information Science, California Institute of Technology, 1975.

B4. _____. "Concurrent Pascal Report," Information Science, California Institute of Technology, 1975.

B5. _____. "Sequential Pascal Report," Information Science, California Institute of Technology, 1975.

B6. Barbacci, M.R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," Carnegie-Mellon University, 1973.

B7. Brinch-Hansen, P., "Concurrent Programming Concepts," ACM Computing Surveys, Vol. 5, No. 4.

C1. Chu, Y., "Computer Organization and Micro-programming," Prentice-Hall, 1972.

C2. _____. "High-level Language Computer Architecture," Academic Press, 1975.

D1. Dijkstra, E.W., "The Structure of THE Multiprogramming System," Comm. ACM 11, 5, pp. 341-46, May 1968.

D2. _____. "Notes on Structured Programming," published in "Structured Programming," Academic Press, 1972.

D3. _____. "A Discipline of Programming," Prentice-Hall, 1976.

D4. _____. "Cooperating Sequential Processes," Technological University, Eindhoven, The Netherlands, 1965 (reprinted in "Programming Languages," F. Genuys, ed., Academic Press, New York, 1968).

D5. Digital Equipment Corp., "PDP 11/45, Processor Handbook," December 1973.

D6. Digital Equipment Corp., "PDP 11, Peripherals Handbook," December 1973.

E1. Enslow, P.H. Jr., "Multiprocessors and Parallel Processing," John Wiley & Sons, 1974.

F1. Fuller, S.H., Chen, R.C., "The I/O Port Architecture for Computer Modules," Carnegie-Mellon University, 1973.

H1. Horning, J.J., Randell, B., "Process Structuring," ACM Computing Surveys, Vol. 5, pp. 5-30, 1973.

K1. Knuth, D.E., "The Art of Computer Programming," Vol. 1, Addison Wesley, 1968.

M1. Mendelson, E., "Introduction to Mathematical Logic," Van Nostrand Reinhold Co., 1964.

M2. Mesarovic, M.D., Macko D., Takahara Y., "Theory of Hierarchical, Multilevel, Systems," Academic Press, 1970.

M3. Mesarovic, M.D., Takahara Y., "General Systems Theory: Mathematical Foundations," Academic Press, 1976.

M4. Mos Technology, Inc., "MCS6500 Microcomputer Family Hardware Manual," Mos Technology, 1976.

P1. Parnas, D.L., Siewiorek, D.P., "Use of the Concept of Transparency in the Design of Hierarchically structured Systems," Carnegie-Mellon University 1972

R1. Rustin, R., ed., "Computer Networks," Prentice-Hall1972

S1. Simon, H.A., "The Architecture of Complexity," Proc. of the American Philosophical Society, 106,pp.467-482, 1962 (reprinted in "The Sciences of the Artif-icial, " Simon, H.A., MIT Press, 1969.

T1. Tsichritzis, D.C., Bernstein, P.A., "Operating Systems," Academic Press, 1974.

W1. Wijngaarden, A., ed., "Revised Report on the Algorithmic Language ALGOL 68", Springer-Verlag, 1976.

Z1. Zadeh, L.A., Desoer, C.A., "Linear System Theory, the State Space Approach," McGraw-Hill, 1963.