

SMART: A Tool for the Study of the ACM Model of Concurrent Computation

by

Richard Edward Yuknavech

B. S., St. John's University, 1960

A MASTERS THESIS

submitted in partial fulfillment of the

requirements for the degree


MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:


Major Professor

LD
2668
T4
1986
484
c. 2

CONTENTS

1. Chapter 1 - Introduction and Overview	1
1.1 Problem Overview	1
1.2 Summary of Current Research	4
1.2.1 Hardware 4	
1.2.1.1 Taxonomy 4	
1.2.1.2 Processor Size and Granularity 5	
1.2.1.3 Current Projects 5	
1.2.2 Theoretical Considerations 10	
1.2.3 Software and Languages 13	
1.3 Intent and Overview of This Document	15
2. Chapter 2 - The ACM Model	16
2.1 An Overview of the ACM Model	16
2.2 The ACM Object	17
2.2.1 The First Component of an Object: Designator 17	
2.2.2 The Second Component of an Object: Attributes 18	
2.2.3 The Third Component of an Object: Representation 18	
2.2.4 The Fourth Component of an Object: Corporality 18	
2.2.5 The Fifth Component of an Object: Value 19	
2.3 Actions and Action Requests in the ACM Model	20
2.3.1 Actions 20	
2.3.2 Action Requests 20	
2.3.3 Action Detailing 20	
2.3.4 Stimulation and Termination Conditionals 21	
2.4 Constructs	23
2.4.1 Alternation 23	
2.4.2 Cascation 23	
2.4.3 Repetition 24	
2.4.4 Iteration 25	
2.5 States of a Request	26
2.6 Partialing	27
2.7 Universality and Determinacy of the Model	27
3. Chapter 3 - Proposed Extensions to the ACM Model	28
3.1 A Problem With the Definition of States of a Request	28
3.2 A Proposed Extension: the FAIL() Pre-defined Action	32
3.3 A Proposed Extension: the S() and F() Builtin Predicates	33
3.4 The Case for the Default Case	34
3.5 A Proposed Extension: the Concurrent Alternative Construct	36
4. Chapter 4 - The Implementation	40
4.1 The SMART Language	40
4.1.1 Implemented and Omitted Features 40	
4.1.2 Syntax 41	
4.1.3 Informal Semantics 42	
4.1.3.1 Object Declarations 42	
4.1.3.2 The Semantics of Dynamic Objects 43	

4.2 Implementation Details	46
4.2.1 Compiler 46	
4.2.2 Node Modules 46	
4.2.2.1 Node Manager 47	
4.2.2.2 Terminal Driver 47	
4.2.2.3 Scheduler 47	
4.2.2.4 Environment Handler 48	
4.2.2.5 Network Interface 50	
4.2.2.6 Atomic Action Library 51	
5. Chapter 5 - Conclusions	53
5.1 Results of this Research	53
5.2 Future Directions	54
Bibliography	57
Appendix A - SMART BNF	62
Appendix B - SMART User Notes	65

LIST OF FIGURES

Figure 1.1. Control Graph for Serial Computation	7
Figure 1.2. A Dataflow Graph	8
Figure 2.1. Illustrative Syntax	22
Figure 4.1. The SMART Action Request Syntax	41
Figure 4.2. SMART Data Object Declaration	42
Figure 4.3. Non-Determinacy and Dynamic Objects	44

Acknowledgements

The actual prototype implementation is the result of a cooperative effort on the part of R. W. Fish, T. E. Shirley, and myself. Ed Shirley, in a programming *tour de force*, produced the entire compiler, the virtual machine loader, and a thesis in a single summer session. Bob Fish, in an equally heroic effort, did the environment handler, the engine which, fueled by Ed's compiler, makes the whole thing go. I did bits and pieces of grunt work.

I am also grateful to Bob and Ed for their willingness to review my ideas, and find the holes in them. I am, however, a little dismayed by the great glee with which they pointed those out. I only hope I gave as well as I got (in both senses).

"There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.

There is another which states that this has already happened."

- Douglas Adams

1. Chapter 1 - Introduction and Overview

1.1 Problem Overview

Every electronic computing device ever built has shared one unfortunate quality with each of its fellows: it is simply not fast enough. The one constant component in the history of computers has been the unremitting demand for ever faster processing. At one time the hunger for raw computational speed may have been the exclusive property of the scientific community: the need for manipulating huge and complex mathematical models and for performing massive data reduction and analysis inspired the creation of some awesome "number crunchers". In more recent times, computer aided design and engineering tools have become popular in many industries. Logic, circuit and physical system simulations are no less demanding on a processor. In the present, the growing glamour of expert systems, robotics, natural language processing and the entire artificial intelligence field in general will surely result in increasing rather than diminishing demand for processing speed.

Historically these demands for more speed have been met primarily by advances in processor hardware technology. The charts of raw processor speed as a function of time are well known and are quite impressive. The history of the price/performance ratio is even more so.

Regrettably the upward trend of processor speeds is slowing, and in fact must someday stop. Electronic computing was not very old when the transmission of an electrical signal through a conductor (at nearly the speed of light) became a practical concern. The development of VLSI

technology reduced, and is still reducing, the spatial separation of elements within a processor, but this path cannot be followed indefinitely. Improvements in this direction must certainly end when the width of the conducting paths within a chip become comparable to the effective radius of the electron itself. More likely the effective limit will be reached sooner, in the form of the problems involved in removing heat from such densely packed elements. Von Neumann machines must also contend always with the classic von Neumann bottleneck, the finite time taken for the necessary communication between the processor and its memory.

Granting the assumption of an upper limit to the power of a single processor, one promising course would be to use multiple processors, working in parallel, to effect a single large computation. Researchers are currently pursuing many variations on this theme. Some architectures consist of essentially von Neumann machines linked by some variant of message passing, or by the sharing of a common memory or pool of memories. One of the most active areas involves a departure from the von Neumann model, a conceptually memoryless set of processors called a Dataflow machine. Common to all parallel processor designs is the concept of *granularity*, a measure of the size and number of the pieces into which the computation can be subdivided, with each piece potentially eligible for parallel execution.

The potential for parallel architectures to provide performance improvement is of course predicated on the tacit assumption that exploitable parallelism or concurrency exists in computations. That is to say that at most times there exist, and can be identified, some number of pieces (grains) of the computation that can be executed simultaneously on separate processor units without impairing the validity of the computation. Investigations have indicated [Alm85] that such concurrency exists, particularly in the types of processor intensive computations cited above. Design and simulation applications should typically offer levels of concurrency in the hundreds. Artificial intelligence and complex scientific applications might well enjoy

concurrency levels in the thousands. Huge potential benefits apparently are languishing about waiting to be realized.

It is, as hinted at above, not sufficient that potential concurrency exist in a computation. It must also be identified before it can be exploited. That particular problem is the target of a great many research efforts, including this one.

The retrospective perception of the deficiencies in the von Neumann architecture should not be interpreted as a condemnation of that architecture. Given the technology of its day, it was a monumental achievement, and it has served well for three decades. Perhaps too well. During these years, the von Neumann computer was seen not as *a kind of* computer, but as *the* computer. Virtually all existing programming languages (excepting the AI languages and a handful of other "strange" things) were shaped by the von Neumann model. They are reifications in software of the von Neumann architecture, and the human mind is shaped by its tools almost as much as the tools are shaped by the mind.

The problem is not that architectures capable of exploiting parallelism do not exist. They do, albeit in a form that will appear quaint and primitive in a few years. Nor is it the fact that current languages hinder, if not totally prevent, the explicit expression of concurrency¹. The real problem is that the minds of most of us have been so canalized by our programming languages that we seldom even think in terms of concurrency.

What is needed are languages that not only allow the natural expression of concurrency, but actively encourage it, or even force it.

1. concurrency-oriented extensions to languages that are intrinsically sequential and imperative are just a step in the right direction.

1.2 Summary of Current Research

Increasing the raw speed of the conventional serial processor by means of incremental technological improvements is reaching the point of diminishing returns. A large part of the community perceives massively parallel processor architectures as the most promising, if not the only, source of large performance gains in the short to medium term. It is hardly surprising that this field is currently one of intense activity.

1.2.1 Hardware The development of VLSI technology was one of the factors that extended the useful life of the classic von Neumann machine. That technology also paved the way for its replacement, for it is VLSI that made feasible the construction of machines consisting of large numbers of small cooperating processors at reasonable cost [Gaj83, Gaj85, et al.]. In this section we attempt a brief overview of the many developments in this area.

1.2.1.1 Taxonomy As if to underscore the diversity of current architectural efforts, several researchers have taken the time to produce taxonomies, systematic classifications, of computer architectures. Flynn [Fly79] introduced some terms that are now routinely used by most authors in the field:

SISD "Single Instruction (stream) Single Data (stream)". This represents the classic von Neumann architecture.

SIMD "Single Instruction Multiple Data" is typical of array processors, where the same operation is performed simultaneously on all elements of an array.

MIMD "Multiple Instruction Multiple Data" encompasses all architectures that consist of interconnected processors. The "dataflow" machines are members of this class.

For completeness, some have added MISD (Multiple Instruction Single Data) to describe pipelined machines.

Schwartz [Sch83] produced a taxonomy that first divided the architectures into two groups (fine and coarse grained) based on the size of the individual processors. These two groups were then

subdivided according to the type of processor interconnection, such as shared memory, message passing, etc.

1.2.1.2 Processor Size and Granularity Two of the variables that may be manipulated by machine designers are the size of the individual processing elements and the granularity of the architecture. Granularity is the measure of the size of the piece of a computation that is given, as it were, as a unit to an individual processor. There tends to be an inverse relation between the power of the processing unit and the fineness of the granularity that is implemented.

The method of interconnection of the processors is important in choosing a granularity, as it obviously makes no sense to expend the equivalent of many instructions to allow two useful instructions to execute in parallel. On the other side of the coin, a very coarse grain size will reduce the processor interconnection overhead, but may result in wasting potential concurrency that exists at finer levels. Some designs attempt to exploit both fine and coarse granularity. These ideas will be illustrated in the next section.

1.2.1.3 Current Projects Technically, a set of small computers on a local area network (e.g., an ETHERNET or a token ring) is an example of an MIMD architecture whose processors communicate by message passing. This configuration is typically used more for the purpose of sharing expensive resources such as large disk devices or sophisticated plotters, than for the gaining of computational speed.

Tightly-coupled multiprocessors, including attached processor configurations such as AT&T's 3B20A or IBM's 370/168MP, neither of which are new, are examples of MIMD architectures with communication by shared memory. These systems are characterized by a small number (two) of powerful processors. Granularity is coarse, typically with a computation running on a processor until it is pre-empted or until it relinquishes voluntarily. When the task is able to

resume, it is dispatched on the next available processor. The opportunity for concurrency must usually be explicitly specified by the programmer. A UNIX² programmer, for example, could realize that two related activities can execute concurrently without damaging each other, and would *fork* another process to run in parallel.

The Cedar machine being developed at the University of Illinois [Gaj83] consists of m Processor Clusters (PC) each of which contains n processors and n local memories. Each processor in a PC can access any of the memories through a switch, but each has one memory to which it has direct access at higher speed. Each can also directly access a global memory. Work is spread over the collection of PC's by a single Global Control Unit. The designers are targeting a top-end system with 1024 processors, each capable of 10 million floating-point operations per second (10 megaflops), which they predict will realize an effective speed of several *gigaflops*. The scheduling and partitioning algorithm for Cedar will be discussed in a later section.

Cm* [Hib78], being developed at Carnegie-Mellon, uses an LSI-11 minicomputer as the processor element. Each processor has access to its own local memory through a switch. The combination of processor, switch and memory is known as a *computer module*, or *Cm*. Cm's are grouped into *clusters*, and finally clusters are grouped to form the machine. There is a hierarchy of memory access: local, intra-cluster and inter-cluster. Accesses to local memory are fastest, to memories in the same cluster next, and to memories in other clusters the slowest. The system attempts to realize both coarse and fine grained parallelism. At the higher level, the computation is divided into "activities". One Cm is responsible for executing each activity, and is called a *master*. The master directly executes parts of the activity and "defers" other parts,

2. UNIX is a registered trademark of AT&T

which are executed concurrently on other (*slave*) processors. The designation of a processor as master or slave is purely dynamic - a master is a Cm that happens to be executing an activity at some instant. More about the Cm* software appears in a later section.

IBM's Research Parallel Processor Prototype (RP3) is designed to contain up to 512 proprietary microprocessors [Pfi85]. This is in contrast to Cedar and Cm* which both use "off the shelf" processing components. RP3 will use both the shared memory and message-passing methods of processor interconnection, has a cache-local-global memory hierarchy and is expected to realize about 0.8 gigaflops.

At the individual processor element level, the architectures described above are not very different from the traditional von Neumann model. There is a great deal of work being done with a class of machines that represent a radical departure from that architecture, the *dataflow* machines.

The operation of any program can be depicted as a control graph such as the one in Figure 1.1, which illustrates a typical control graph for a serial (non-concurrent, von Neumann) computation.

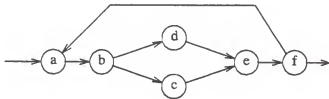


Figure 1.1. Control Graph for Serial Computation

The choice of path along this graph is controlled by a special register within the processor, called the location counter³. By default, the location counter is incremented by one unit with each

instruction execution, but the program has a limited explicit control over it by means of jump instructions. If the control graph of Figure 1.1 is transformed by changing the circular nodes to rectangles and diamonds as appropriate, the homely figure of the "Program Flowchart", familiar to us all, will emerge. It is important to note that at node *b* a decision is made, and for any single pass through this graph control will go to either node *c* or node *d*, but *not both*.

In Figure 1.2, we see an entirely different sort of graph, truly two-dimensional as contrasted to the linear form of the serial graph.

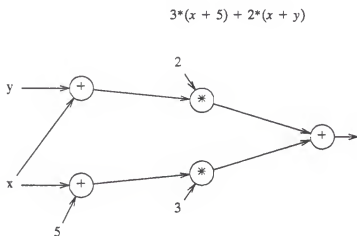


Figure 1.2. A Dataflow Graph

This type of graph may be considered the "machine language" of the dataflow computer. In the serial model graph, the nodes represented data operations and the arcs indicated the flow of control from operation to operation. In the graph of Figure 1.2 the arcs represent the flow of the *data* from operation to operation, hence the name "dataflow graph" for this type of graph,

3. or program counter, or program address

and "dataflow computer" for the machine that interprets it.

In a true dataflow machine an operation is performed whenever all of its data are available. Usually understood as implicit in the notion of dataflow machine is a micro-fine granularity, as fine as a single machine instruction, with a large number of small processors. The "+" and "*" operations shown in Figure 1.2 are doubtless realizable as single instructions in whatever processing element underlies the architecture. In the figure, we can expect that x will be added to the constant "5" on one processor, and will be added to y on another concurrently. As soon as either of these operations has produced its result, the corresponding multiplication will be performed, possibly both concurrently. The final "+" will perform be delayed until both multiplications have completed.

Those acquainted with Petri Nets [Pet77] will have noticed the close resemblance between the appearance of data at the inputs to a node, causing that node to execute, and the appearance of tokens at input places to a transition, causing that transition to "fire". This resemblance is not accidental. Indeed, many dataflow descriptions even speak of the data moving along the arcs as "tokens".

In the mode of operation just described, an operation is performed whenever all of its data are available. This is called *data drive*. There exists the possibility of some unnecessary computation, the evaluation of results that are not actually used by virtue of subsequent conditional execution. There is another variant in dataflow technology called *demand drive* in which the graph is processed in "reverse" order. The machine determines what immediate inputs are needed for its final result, and what inputs are needed to compute *them*, and so on. Only the computations actually needed to produce the final output are done.

Highly visible work in dataflow hardware and software technology has been done in the United

States at MIT [Den80] under J. B. Dennis, the acknowledged pioneer in the field, and at the University of California by Arvind [Arv80]; in France at ONERA-CERT [Fra85]; and in England at the University of Manchester [Gur85].

1.2.2 Theoretical Considerations The earliest efforts in parallel processing to gain performance improvement used what we now call the shared memory paradigm. Typically the machine consisted of two von Neumann processors sharing a single large memory ("tightly coupled multiprocessors"). The memory contained a single copy of the code and data both of running computations and of the operating system.

There was an obvious need in such architectures to discipline and control the access to global data structures in memory. The first development in that area was Dijkstra's concept of the semaphore [Dij68]. The semaphore enabled the program to serialize access to global data on the part of the two processors, but did so at a very low level.

Hoare [Hoa74], building on Dijkstra's suggestion of a higher-level program unit to coordinate shared data access, formalized the notion of *monitors*, and Brinch Hansen [Han75] produced the programming language Concurrent Pascal, which included monitors as a language feature. These developments did much to ease the trauma that always accompanies any non-trivial programming effort in shared memory architectures.

Another attack on the problem of concurrent access to shared global data is to eliminate the use of such data, and to allow cooperating tasks to communicate by *message passing*. Hoare, in "Communicating Sequential Processes" [Hoa78], has such processes interacting only through input/output constructs. Data abstraction models encapsulate data, and procedures to manipulate that data, into a single unit (CLU [Lis77], Alphas [Wul76]). Whatever the underlying physical reality, such models conceptually use message passing as the communication

mechanism.

In the world of dataflow architectures, given that machines exist or will exist that can efficiently interpret and execute a two-dimensional dataflow graph, the only (!) remaining problem is the production of that graph. Obviously a dataflow graph can be produced explicitly by a programmer, but for a computation of any useful size this is hardly practical. It is necessary to provide languages with a linear, human form and language translators that can convert that form into the best achievable dataflow graph.

Dataflow language researchers are unanimous in declaring that two features present in most conventional languages enormously complicate this task. These unfriendly features are the presence of *side-effects* and the absence of *single assignment*.

In the single assignment model, once the value of a data object exists, it may never again be changed. In the classic von Neumann world, the statement $a:=b+c$ implies that somewhere there exists a storage cell uniquely identified by the label "a", and that the program is at liberty to compute values and store them in that cell whenever, and as often as, it pleases. The damage this does to a dataflow philosophy is easy to see. Once "a" has a value, all operator nodes in the graph that have "a" as an input are eligible for concurrent asynchronous execution⁴. If the program at some later time, but prior to the actual execution of all such nodes, should assign a new value to "a", chaos will obviously result.

When speaking of single-assignment languages, one usually does not say "assign a value to a variable", but rather "bind a value to a symbol". Ackerman, in describing *definitional* languages [Ack79], points out that the statement $a:=b+c$ should be thought of as a mathematical

4. assuming of course that all of their other inputs likewise have values.

equation, along with the other such statements in a program. With this interpretation, wherever and *whenever* the symbol "a" appears in an expression, the expression " $b+c$ " may be substituted. The statement is therefore not an assignment, but rather a *definition*. By extension, the statement $a:=d+e$ anywhere within the same scope becomes a *redefinition* of "a", and is therefore in error. The value to dataflow languages of this interpretation is quite clear.

The other property much beloved by dataflow enthusiasts, freedom from side-effects, can be found in the class of languages called *functional languages*. The name springs from the fact that these languages operate by *applying* functions to *values* according to the strict mathematical definition of "function" [Bac78, Kog85, et al.]. Under this interpretation, the same function can be applied to the same values *at any time* and produce the same resultant value.

This is in contrast to the style of traditional von Neumann languages, wherein of necessity statements and procedures must compute by changing the values of local and global state variables. Automatic re-ordering of the relative time sequence of statements is difficult; of procedures, almost impossible.

One problem faced by definitional languages involves statements like: $j:=j+1$, to which Ackerman [*op.cit.*] applies the epithet "mathematical abomination", but concedes that they are an apparent necessity within iterative constructs. Although such a statement, interpreted either as a definition or as a mathematical equation, is an absurdity, definitional languages do deal with them. In languages such as VAL, ID or LUCID such redefinitions are permitted only in the bindings of loop variables, and the semantic description of exactly when the redefinition takes effect is precise.

A similar problem exists in functional languages when attempting to manipulate real world objects that *do* possess the notion of history. Halstead [Hal85], in describing the deliberate

inclusion of side effects in the language Multilisp, offers database update as one such. Kogge [Kog85], citing the work of Keller and Lindstrom [Kel85], asserts that the problem is solvable and being solved, and that functional style will in fact solve many of the difficult problems exhibited by concurrently-accessed databases.

1.2.3 Software and Languages Each of the hardware development efforts outlined in Section 1.2.1.3 has an associated software effort. Here we present a necessarily brief description of each in the hope that the reader will be able to gain at least a feel for the diversity (and similarity) of current efforts.

Parafraze [Gaj83] is the language system being developed for the Cedar machine. The system will accept both standard FORTRAN and a FORTRAN extended with concurrency constructs. The intent is to automatically discover parallelism in the input source program at coarse, medium and fine levels and to produce what is essentially a macro-dataflow graph for the run-time system to execute.

The language for the Cm* machine [Hib78] is an Algol 68 subset with extensions providing for the explicit expression of coarse-grained parallelism. This specification defines the "activities" described earlier as the unit of work dispatched to a "master" Cm. The software hopes to automatically detect finer-grained parallelism within each activity and spread that work concurrently across "sub-masters" and "slaves".

Multilisp [Hal85] is a Lisp dialect that includes deliberate side-effects, shared memory and explicit parallelism. This would appear to be an odd mix of features for a language that is descended from one that in its pure form is entirely functional, but the authors present a logical defense of this design. First, they were concerned with the ability of a purely functional language to work with objects that inherently contained a sense of history (e.g. databases). Thus

they permitted side-effects. Second, shared memory eases the task of exploiting lower-level parallelism. Finally, the presence of side-effects greatly complicates the automatic detection of coarse-grained parallelism, thus explicit expression of same was provided.

VAL [McG82] is a small research dataflow language developed at MIT. Its intent is to discover the maximum fine-grained concurrency available in any computation without requiring (indeed not permitting) explicit specification of that concurrency. To this end, it is a value-oriented (single assignment) purely functional (no side effects) language.

For the ID language, Arvind [Arv80] devised a *tagged token* scheme to address a problem with iterative constructs. Earlier dataflow architectures allowed only one data "token" to exist on an arc at any one time. This can result in the loss of otherwise realizable concurrency during an iteration. The tagged token architecture permits multiple tokens to reside on an input arc, but "tags" each one differently. These distinctive tags are often called "color". A node within an iteration then does not always execute whenever it has tokens present at all of its inputs, for there must be a full set of tokens *of matching color*. Thus any instance of an operation within an iteration always works with a set of values belonging to the same iteration number. The Manchester prototype dataflow machine [Gur85] also uses this technique.

In France at ONERA-CERT, work is in progress with two languages, LAU and LESTAP. LAU [Com78], one of the earliest dataflow languages, is an object oriented single assignment language. LESTAP [Fra85] is a FORTRAN 77 precompiler that provides for explicit specification of coarse concurrency and produces output to execute on a system of four loosely-coupled array processors.

1.3 Intent and Overview of This Document

The preceding subsections of this chapter have provided background on the subject of parallel or concurrent processing. The purpose of this project is to describe a primitive but working prototype of a language based on the ACM model [Ung78] of concurrent computation. This model will be described in detail in Chapter 2.

Chapter 3 describes problems with the model that surfaced during the intensive study required by an implementation. Since the activities involved in designing such an implementation are in a sense practical experience with the model, some extensions to the model suggested themselves. These are also developed in Chapter 3.

Chapter 4 describes the implementation, including

- The formal syntax of the language.
- A very informal description of its semantics.
- A description of the run-time support system developed for it.

Chapter 5 provides a summary and review along with suggested directions for future development of the prototype and a reinforcement of those problems mentioned in Chapter 3 that were not resolved there.

2. Chapter 2 - The ACM Model

2.1 An Overview of the ACM Model

A *Concurrent Model (ACM)* [Ung78] is a model of computation that is intrinsically concurrent. That is, the time sequence of actual execution of actions within the model is independent of the physical order of appearance of requests for those actions in the "source code", which nonetheless may be expressed in a familiar linear syntax. The actual sequence of execution is based on *data-drive*, which causes an action to be eligible for execution whenever all of its materials are available.

The model also incorporates the notion of explicit control over execution, effected by optional explicit *stimulation* and *termination* conditions. These are sufficient to provide the effect of the recognized "structured programming" control constructs: alternation, caseation, and "looping". The term "looping" encompasses both iteration and repetition, two terms between which the model makes a deliberate distinction which is meaningless in the realm of sequential computation but critically important in the concurrent world.

Everything exists in the model as an *object*. The model demands that once a value is bound to a particular object, that value never changes⁵, which is to say the model incorporates the principle of *single assignment*.

The model provides for the formation of structures by defining the process of *aggregation*, in such a way as to result in the principle of *data abstraction*.

5. With one exception, which will be described in due course.

Actions within the model are a particular type of object, and a program in the model is a set of *requests* for those actions. Actions can be *detailed*. That is, an action can be represented as a set of requests for "finer" actions (a *request set*), thus providing for modular decomposition.

2.2 The ACM Object

An object in ACM is formally represented as a five-tuple: (d, a, r, c, v) , which stands for *designator*, *attribute*, *representation*, *corporality*, and *value*. These will each be described in the following subsections.

2.2.1 The First Component of an Object: Designator The designator is a four-tuple: (c, u, i, a) , *context*, *user name*, *instance*, and *alias*.

The context component of the designator is a (possibly empty) list of names that provides for the identification of the object outside of its context of creation. An example might be: "net3.proc1.job6".

The user name of an object is a simple name, or possibly a list of such names, that uniquely identify that object within its context of creation.

The instance component is further broken down into *spatial position* (think of array indices), *chronological identity* based on a specifiable "clock", and a *sequence number* (represented as, e.g., "name..3").

The concept of alias is mentioned in the model, but is not further developed at this time.

The model defines exactly what is meant when only a part of the designator is used to reference an object. Of particular interest is the notion of omitting some or all of the spatial coordinates when referencing an object whose designator includes them. This provides the ability to

reference all or part of an array. Also of interest is a reference without an explicit sequence number to an object that has a sequence number associated with it. Defined as retrieving the highest-numbered object with that user name, this provides the ability for an object to "appear" to change values during an iteration.

2.2.2 The Second Component of an Object: Attributes This component provides the notions of the type of value that can be associated with the object, the internal structure of an object, and its possible relation to other objects (e.g. an ordered set).

2.2.3 The Third Component of an Object: Representation This component deals with the physical realization of the object, such as location, encoding, etc. Representation is not further developed in the model as yet, as it is primarily an issue of implementation.

2.2.4 The Fourth Component of an Object: Corporality Corporality is a four-tuple, (l, p, r, e) : longevity, location (place), replication, and authorization.

In the model, longevity is the most pervasive subcomponent of corporality. It can have one of four values:

fixed Objects with longevity "fixed" exist prior to the model and can never be changed. Such things as the set of decimal digits or the letters of the alphabet are considered "fixed".

static these objects are created within the model and are never changed. An example would be a "static constant".

dynamic these objects are created within the model and can appear to change value. The appearance of change is effected by the creation of a new incarnation of the object, with a different sequence number, (hence a different designator and therefore a

different object) and by the fact that a reference using only the simple name of the object (the μ subcomponent of the designator) references the latest such incarnation. By including an *absolute* or *relative* sequence number in a reference (such as $x..5$ or $x..-1$) the values of previous incarnations of the object can be obtained. These properties are most important in the area of iteration. Most "variables" within the model are dynamic.

fluid these objects can really change value (no new incarnation is created, no previous incarnations exist) at any time between uses. An example might be the temperature of some physical process being monitored.

Location refers to a physical location of a value, and is therefore strictly an implementation concern. It is not developed in the abstract model.

Replication includes an indication of the availability of the object, its copyability and the number of copies of the object that exist in the model.

Authorization provides the concepts of security and protection of an object.

2.2.5 The Fifth Component of an Object: Value The value component of an object is in a sense the entire reason for the existence of the object. The value of a data object is usually created by some action within the model. The value of an action object is created through the process of *action detailing* described in the next section.

The components of an object need not be bound, or come into existence, simultaneously. An object is considered to exist as soon as its designator exists, and can be uniquely referenced by its designator. The value component must be the last component to be bound, and once that happens the object is said to be *complete*. Except for the value component of an object with longevity fluid, these bindings are permanent.

2.3 Actions and Action Requests in the ACM Model

2.3.1 Actions An action is an object that specifies some operation to be performed on other objects. In its simplest form, an action is a three-tuple, (m, a, r) , where a is the designator of the action, and m and r are the *material list* and *result list* respectively. The material list is a list of designators of material objects which will be used by the action to produce other (resultant) objects. The result list is the list of designators of those resultant objects.

2.3.2 Action Requests A *request* for an action is the basic "executable" entity within the model. Abstractly it is represented by the same (m, a, r) three-tuple. In examples throughout this paper we will use the following, more human, syntax:

`egorps(m1, m2, ... ; r1, r2, ...);`

At this point we don't know what the action "egorps" does. We do know that it uses the objects m_1, m_2, \dots as "input" material and through some transformation produces the value components of the objects r_1, r_2, \dots . In later examples in this paper, if the effect of such an action is important, that effect will be specified. Other actions used in examples will have effects which are intuitively obvious from the action name, such as "add(a, b; c)".

2.3.3 Action Detailing An action may be *detailed* by attaching to it a value that consists of a set of *requests* for more primitive actions. These requests are said to constitute a *request set*. Individual requests in that request set may be requests for actions that are themselves detailed. Ultimately a level of detail must be reached where actions can not be further refined. Where this happens depends on what is being modeled. In a model of a computation, for example, "add(a, b; c)" would be considered primitive. For a model of the computer itself, this action might be further detailed to the level of the propagation of bits in an adder.

In examples to follow, we will use the following convention to indicate detailing:

```
action( m; r )
{
    act1( m1; r1 );
    act2( m2; r2 );
    . . .
}
```

where the materials and results of the action and of the requests detailing the action, as always, can be comma separated lists of objects.

There are certain restrictions on what can appear in the material and result lists of the requests detailing an action. In particular, an object in the material list of any request in the set will either be in the *material* list of the detailed action, or in the *result* list of exactly one other action request in the set. The first condition is analogous to a formal parameter in a function, while the second is something like an "intermediate result". Each object in the result list of the detailed action must appear in the result list of some request in the set.

2.3.4 Stimulation and Termination Conditionals With actions and requests defined as above, the order of execution of requests can only be determined by *data drive*. That is, a request is eligible for execution if and only if the action itself is available, all of the objects in its (the request's) materials list are complete and available, and the user is authorized to use the objects.

To provide for conditional execution, and thus for the constructs of alternation, caseation, iteration and repetition, the model actually represents actions as a five-tuple: (s, m, a, r, t) , where s and t are called the *stimulation* and *termination* respectively, and are both boolean expressions. A stimulation or termination appended to an action is an *internal* stimulation or termination. A stimulation or termination appended to an action request is an *external* stimulation or termination. Thus an action is represented as: (s_i, m, a, r, t_i) , and a request as: (s_e, m, a, r, t_e) .

Stimulation and termination conditions are optional in any action or request. A null stimulation is considered always TRUE, and a null termination is always FALSE. The *effect* of a request is defined as: (s, m, a, r, t) where $s = s_e \wedge s_i$ and $t = t_e \vee t_i$. Finally, the effect of a request is eligible for execution if data drive is satisfied **and** s is TRUE **and** t is FALSE.

To give some examples, we must first illustrate the syntax that we will use:

```
For an action:
  actname ( [  $s_e$  ] m1,...; r1,... [  $t_i$  ] )
and for a request:
  [  $s_e$  ] actname ( m1,...; r1,... ) [  $t_e$  ] ;
```

Figure 2.1. Illustrative Syntax

The square brackets ([]) above are not meta-symbols. The stimulation and termination are boolean expressions enclosed in square brackets.

Now consider an action and a request for it:

```
act1 ( [  $n > 0$  ] n, nums; mean )
[  $n < 1000$  ] act1 ( n, scores; average ) ;
```

Here the action presumably sums the first "n" elements of an array and divides by "n" to produce the arithmetic mean of those elements. The programmer apparently doesn't want to bother if there are too many elements, and the author of the action isn't at all interested in dividing by zero. Since the effective stimulation is the logical **and** of the external and internal stimulations, the request will be considered for execution only if $1 \leq n \leq 999$.

2.4 Constructs

We are now prepared to examine the way in which the model abstractly represents the constructs of alternation, caseation, repetition and iteration.

2.4.1 Alternation An alternation consists of two requests in a request set of the form:

```
[ s1 ] act1 ( m1; r1 );  
[ s2 ] act2 ( m2; r2 );
```

where both stimulation conditionals *must* be present, and $s1 = \text{not } s2$. Then one and only one of the requests will execute, yielding the exact effect of the if..then..else construct. A trivial example is an action that determines the maximum of a pair of numbers:

```
max (a, b; c)  
{  
  [a > b] assign (a; c);  
  [a <= b] assign (b; c);  
}
```

2.4.2 Caseation Although the effect of caseation can always be duplicated by a series of nested binary alternations, the model, like most languages, formally recognizes a more convenient notation. Caseation is defined as a set of n requests in a detail of the form:

```
[  $s_i$  ] act $i$  (  $m_i$  ;  $r_i$  );
```

where no s_i is null, and where the s_i 's are mutually exclusive. That is, no two can be TRUE for any possible value of the materials.

The following example is an example of a simple caseation:

```
case ( x; y )
{
  [ x < 0 ]          'assign ( 0; y ) ;
  [ (x >= 0) & (x < 50) ] mult ( x, 4; y ) ;
  [ x >= 50 ]       div ( x, 2; y ) ;
}
```

2.4.3 *Repetition* is the term used in the model to describe a succinct notation that conveniently represents the SIMD operation typically found in vector or array processors. The rigorous definition of the conditions under which the model will recognize and effect repetition is lengthy and complex, and will not be repeated here. Instead, we will address only the simplest case of repetition, the starting point of the model's complete definition, to indicate the flavor of the construct.

Given a set of requests $(s_i, m_i, a_i, r_i, t_i), i = 1, \dots, n$, where for all $i, j = 1, \dots, n$ the conditions $a_i = a_j, s_i = s_j, t_i = t_j$ all hold, then *repeated requests* are said to exist.

The *simple repetition* construct is a single request (s, m, a, r, t) where there is at least one object, o , in the materials list and one object, p , in the results list that have multiple values (*i.e.*, arrays), and

- the number of values in each $o, |o|$, is equal to the number of values in each $p, |p|$, and
- The computation could have otherwise been expressed as $|o|$ requests each expressed in terms on *one* value for each o and each p and
- All other objects in m and r have one value.

For example, if a, b and c are three arrays of the same size, and if a computation could have been written as:

```
add ( a(1), b(1); c(1) );
add ( a(2), b(2); c(2) );
. . .
add ( a(n), b(n); c(n) );
```

where n is the number of elements in the arrays

then the model will recognize the simple repetitive construct:

```
add ( a, b; c);
```

and generate n independent requests to perform the computation.

The "complications" mentioned above arise from the need to rigorously define the general case of repetition in the presence of aggregations and other "nasty" things. The interested reader is referred to [Ung78a] for full treatment of this topic.

2.4.4 *Iteration* is a repeated action in which some sequence of values is computed and where each of the values depends on one or more of the values computed earlier in the sequence. Thus where the n computations of the Pascal loop:

```
for i := 1 to n do  
  c [ i ] := a [ i ] + b [ i ];
```

obviously could all be done simultaneously, the computations of:

```
a [ 1 ] := 1;  
a [ 2 ] := 1;  
for i := 3 to n do  
  a [ i ] := a [ i - 2 ] + a [ i - 1 ];
```

just as obviously could not.

In the model, iteration *always* involves a request for an action that contains an internal termination condition. It usually also involves the production of several generations (new incarnations) of one or more dynamic variables. The internal termination condition uses the value of an incarnation of one such variable. The formal definition specifies that the termination condition is evaluated *before* the first execution of the action and again after each execution of the action. If the termination conditional expression evaluates to TRUE, the action

(and the iteration) is terminated. If FALSE, the action is eligible for (re)execution. This gives the semantics of iteration with test before execution, similar to Pascal's "while ... do ...", as opposed to the Pascal "repeat ... until ...". In other words, *zero or more* executions of the loop will be done.

The best simple example of this is the ever-popular Fibonacci number series, which is produced in the model thus:

```
(x is a dynamic object)
assign ( 1; x..0 );
assign ( 1; x..1 );
add ( x..-1, x..+0; x..+1 [ x..0 > n ] );
```

2.5 States of a Request

In the model an action request can be in one of three states: *idle*, *enabled*, or *disabled*. A request enters the idle state in one of two ways. When a request set is enabled, all of the individual requests in it are put into the idle state. Requests also enter the idle state if they are within an iteration, whenever the internal termination condition that will stop the iteration is tested and is false.

A request in the idle state moves to the enabled state whenever and as soon as its materials are available and its stimulation condition is true. An enabled request is eligible for execution whenever a processor is available.

A request moves to the disabled state from either the idle or enabled state as soon as its external termination condition becomes true. In this case, it has produced *none* of its results. A request also moves from the enabled state to the disabled state as soon as all of its results exist and are

available. This is the notion of "completion" of a request.

2.6 Partialing

The model also supports the concept of *partialing*, in which a (presumably complex) detailing of an action includes an explicit indication that the strict application of the data drive principle may be relaxed when considering any request for that action. Specifically, one or more items in the materials list of the action are marked as unnecessary for initiating activation of the request. The action can be started whenever all objects in the material list that are *not* so marked are available, and will execute all actions in the detail that can be executed with the materials that *are* available. Execution is then suspended until more of the material objects become available. Any request for that action may optionally negate partialing for that particular request.

2.7 Universality and Determinacy of the Model

In [Ung78a], the model is proved to be *universal*. The proof is based on the already proven universality of the Turing Machine model, and is done by proving that the model can simulate any Turing Machine.

The full model supports non-deterministic computation. By defining a number of restrictions on constructs and on the material and result lists of action details, [Ung78a] defines a subset of the model, called the "core". This subset is then proved to be determinate.

3. Chapter 3 - Proposed Extensions to the ACM Model

During the design of this prototype, some weaknesses or omissions were discovered in the original model. Also some potentially useful extensions to the model were discovered. These are discussed in the following sections.

3.1 A Problem With the Definition of States of a Request

Chapter 4, Section 4.2 of [Ung78a] deals with definitions of the possible states of an effective action request and with the possible "life cycles" of a request. The following paragraphs are a restatement of those definitions, partially paraphrased. The definition labels are the same as those in the original paper to facilitate cross-referencing.

Definition 4.2-2: The state of a request is one of:

1. Idle- The effect of the request is not eligible for activation as determined by data drive, external stimulation condition or internal termination condition.
2. Enabled- The effect of the request is eligible for activation.
3. Disabled- The effect of the request has been considered for activation and the request executed to completion or was not enabled.

Definition 4.2-3: An effective request R transitions to the *idle* state ("is idled") when a request for the action of which R is a part is enabled⁶. Additionally, R is idled if the action is an iteration construct [at the boundary between iterations]. Under no other conditions is R idled.

Definition 4.2-4: An effective request R is enabled if and only if:

1. R is in the idle state, and
2. R 's external stimulation condition is TRUE, its external termination condition is FALSE⁷, and all objects in its material list are available.

6. This would imply that the "outer block" is semantically a request set. No objection.
7. An omitted (null) stimulation condition is always TRUE, an omitted termination condition is always FALSE.

A request is enabled as soon as these conditions are met. Thus when a request set is enabled, all requests in that request set which may initially be enabled are enabled [atomically].

Definition 4.2-5: An effective request R is disabled if and only if:

1. Its external termination condition is TRUE, or
2. R is in the enabled state, its external termination condition is FALSE and every object in its result list exists and is available.

A request is disabled as soon as these conditions are met.

Definition 4.2-6: An effective request is said to *terminate* if it has gone through the state transitions: $idle \Rightarrow enabled \Rightarrow disabled$.

Definition 4.2-7: A *hang-up state* occurs when none of the requests detailing an action is enabled, and at least one of those requests is in the idle state.

[Informally], the life cycle of an effective request is one of:

1. $idle \Rightarrow disabled$ -- "not executed"
2. $idle \Rightarrow enabled$ -- "in execution" or "in hang-up"
3. $idle \Rightarrow enabled \Rightarrow disabled$ -- "terminated"

With these definitions as they are, we are left with several problems in the model. For example, the alternation construct. Assuming the following informal syntax of an action request:

[<ext-stim>] <name>(<mat-list>; <res-list>) [ext-termin.]

then an alternation is a construct of two requests in a request set:

[s1] act1 (m1; r1)
[s2] act2 (m2; r2)

where $s1$ and $s2$ are both non-absent boolean expressions and $s1 = not\ s2$. That is, one and only one of $\{s1, s2\}$ is TRUE, and therefore one and only one of the two action requests will be enabled, assuming that all materials are available. Assume that request to be "act1".

If the request set is "correct", then the remaining requests in the set will be able to produce the entire result list of the set, and the request set will be eligible for termination (by condition (2) of Definition 4.2-5). But by Definition 4.2-3, "act2" was idled when the request set became

enabled, and nothing in Definitions 4.2-4 and 4.2-5 caused it to change to another state. By Definition 4.2-7, a hang-up state exists, and we are therefore forced to conclude that any correct alternation results in a hang-up state.

There are two possible ways to eliminate this problem. The easiest is to change the definition of alternation to require both stimulation and termination conditions:

$$\begin{array}{l} [s1] \text{ act1 (m1; r1) } [s2] \\ [s2] \text{ act2 (m2; r2) } [s1] \end{array}$$

With this formulation, both requests will be icled when the request set is enabled, then (assuming $s1$ is TRUE and $s2$ FALSE) "act1" will be enabled, and "act2" will be disabled (by condition (1) of Definition 4.2-5). The hang-up state will then not be reached.

Although this solution works, there will (probably) be similar problems found elsewhere in the model. A more general solution to that whole class of problems is the addition of a third condition to Definition 4.2-5, which would specify that an effective request transitions to the disabled state from *either* the idle state or the enabled state whenever the request set of which it is a part switches from the enabled state to the disabled state.

The definition of "terminated" in Definition 4.2-6 must also be enhanced if the availability of the objects in an effective action's result list is to be unambiguously determinable. By Definition 4.2-5, an effective action can move from the idle state *or* from the enabled state whenever its external termination condition is (or becomes) TRUE. This is condition (1). An effective request can also move from the enabled state to the disabled state by virtue of having produced all of its results⁸. This is condition (2). If the previous proposal is accepted, there would be a

8. Actually, the definition merely requires that all objects in its result list exist and be available. It does not explicitly prohibit their production by some other concurrent action. This will have ramifications on caseation and other constructs in the full model.

third condition defining a transition from idle or enabled to disabled.

Given that an effective request can reach the disabled state *without* producing any of its results, it is possible for a request set to reach a condition wherein all of the requests in that request set are disabled, but the request set as a whole has not produced all of its results. This condition is not defined in the model.

The proposed modification is to combine Definitions 4.2-5 and 4.2-6. In the process, the concept of "terminated" (currently defined in 4.2-6) would be enhanced to formally distinguish between the conditions "completed" and "failed". In addition, the concept "not executed", which is informally developed in the model, would be formally defined. The suggested new version of Definition 4.2-5 is:

Definition 4.2-5: An effective request R transitions to the disabled state as soon as **any one** of the following conditions becomes true:

1. Its external termination condition is TRUE.
 - a. If R is in the *idle* state at that instant, then R is said to have **not executed**.
 - b. If R is in the *enabled* state at that instant, then R is said to have **failed**.
2. The request set of which R is a member has become disabled.
 - a. If R is in the *idle* state at that instant, then R is said to have **not executed**.
 - b. If R is in the *enabled* state at that instant, then R is said to have **failed**.
3. R is itself a request set which is enabled, and all of the requests within R have reached the disabled state, but at least one object in R 's result list has not been produced. R is said to have **failed**.
4. R is in the enabled state, its external termination condition is FALSE, and all of the objects in its result list have been produced. R is said to have **completed**.

An effective request that has failed or not executed has produced *none* of its results.

Theorem 4.2-5, which deals with the question of whether an effective request has produced its results, then becomes superfluous and can be dispensed with.

It seemed intuitively attractive, for a while, to define a fourth state of an action request: the *initial* state. On more careful reflection it developed that the definition we had in mind for the initial state (assuming "the usual" scoping rules) would only apply to action requests that could not be known in any active context anyway.

3.2 A Proposed Extension: the FAIL() Pre-defined Action

In practical computations or simulations, an algorithm can often determine entirely internally that it is unable to proceed. It must then be able to signal to the outside world that it has not been able to produce useful results. In the model this can always be done by producing an additional result, a boolean, which indicates the validity of the action's "real" result(s). There are times, however, when the production of any of the results of such an action is eminently inconvenient. See, for example, the discussion of the *Concurrent Alternative* construct in a later section of this chapter. In these cases, and for more understandable notation in general, it is suggested that the model incorporate a pre-defined action, *FAIL()* to be used in the request set detailing an action. If the *FAIL()* request is ever enabled (a request for this action would never have a null stimulation condition), its effect would be as follows:

1. The request set would move to the disabled state (from, of course, the enabled state).
2. All other idle or enabled action requests in the request set would be disabled and considered failed or not executed in accordance with condition (2) of the proposed amended version of Definition 4.2-5 described in the previous section. This point is probably moot, since with normal scoping rules these effective actions should be unknown

outside the request set. The statement should be made for the sake of completeness.

3. The request set will not have produced any of its results.
4. The request set will be considered to have *failed*.

3.3 A Proposed Extension: the S() and F() Builtin Predicates

It is often necessary to base the stimulation or termination of an action request on the termination of some other action. It would seem to be always possible to do this by testing the availability of an object in the other action's result list (creating a "dummy" result if necessary), but the notation can become cumbersome. In cases where the other action could fail to produce its results because of an internal failure such as the use of the FAIL() action described above, this communication becomes very difficult indeed.

The proposed solution to this problem involves defining another component of an action request: the *label*. Then in the illustrative syntax used earlier, a labeled action request could look like:

try1: [x > 0] algor1 (x, y, z; a)

Then two built-in predicates would be available: *S(try1)* and *F(try1)*. There is no obvious reason to allow these predicates to appear anywhere outside of external stimulation and termination conditions. The formal definition of these predicates would be:

Definition: The predicates *S(x)* and *F(x)* are evaluatable whenever an action request with a *label* component of *x* exists. If the action request *x* is in either the idle or the enabled state, then both *S(x)* and *F(x)* have the value FALSE. If the action request *x* is disabled, then by Definition 4.2-5 (above), it has either *completed*, *failed* or *not executed*. If *completed*, then *S(x)* is TRUE and *F(x)* is FALSE. If *failed* or *not executed*, then *S(x)* is FALSE and *F(x)* is TRUE.

3.4 The Case for the Default Case

In the original description of the model, Definition 3.2-1 defines the *caseation* construct, and may be stated (using the illustrative syntax) as:

Definition 3.2-1: *Caseation* is a construct of n requests (n finite), given in the detail of an action, of the form:

$$[S_i] \text{ action}_i (m_i; r_i)$$

where $S_i \neq \phi$, for all $i = 1 \dots n$ and
 $(S_i \wedge S_j = \text{TRUE}) \Rightarrow i=j$

This definition guarantees that *at most* one action request will be enabled, but omits the explicit statement of whether or not the disjunction of all of the S_i 's must be a *complete* disjunction. Such a statement, combined with the requirement of mutual exclusion already present, would demand that *exactly one* request in the set be stimulated.

This is not absolutely necessary, since if all S_i 's are FALSE the request set will reach a hang-up condition. The computation will then presumably collapse, which is (probably) a good thing to have happen.

It would be more comforting, however, to incorporate a firm statement into the definition as a binding on potential language developers. A slightly relaxed interpretation of the existing definition would permit the semantics of the C Language version of caseation, which proceeds silently past the construct (with no effect) if no case is selected. This is an abomination.

The proposal is that Definition 3.2-1 be extended to also specify the effect within the model of the enabling of a caseation construct *none* of whose stimulation conditions is TRUE. This effect should be to *fail* the request set.

This naturally forces a discussion of what is usually called a *default* case. We can define this as follows:

Given the predicates $S_1, S_2 \dots S_n$, that are mutually exclusive but do not form a complete disjunction, then the default predicate, S_d , is a predicate such that the predicates $S_1 \dots S_n, S_d$ are mutually exclusive, but do form a complete disjunction.

S_d , of course, can always be formed by:

$$\text{not } S_1 \wedge \text{not } S_2 \wedge \dots \wedge \text{not } S_n$$

which is easily seen to have the desired properties, given that the predicates S_i are mutually exclusive.

In the model, or in any programming language that permits boolean expressions as case selectors, the programmer can always generate S_d mechanically. In a programming language that permits only constant expressions as case selectors, it can be done with a degree of difficulty that depends on the manner in which the S_i 's divide up the domain of the "switch variable". Usually some reasonable combination of enumeration of excluded cases within the caseation and judicious pre-filtering can be found. All of these can be cumbersome, tedious, and error-prone, and negatively affect readability and maintainability.

It is no doubt apparent by this time that the author is in favor of explicit support for a notational shorthand for S_d in programming languages. It should therefore also be added to the model as another strong suggestion to potential language developers.

There are some enormous practical problems involved with this suggestion, especially in the unrestricted model. Definition 3.2-1 does not explicitly require that the set of requests that constitute the caseation be the only requests in the request set. Such a restriction is in fact not necessary. The definition can be interpreted as stating that if there exists some set of requests whose external stimulation conditions satisfy Definition 3.2-1, then a "semantic effect" exists to

which we arbitrarily, but with an eye on the outside world, attach the name "caseation".

At this point, a little "thought experiment" might be instructive. Imagine a program written in the illustrative syntax. As will be seen in Chapter 4, a prototype exists which can compile and execute this program. The program is a single large request set; there is no action detailing. Let this program be punched on a deck of tab cards, one request per card. Assume that there are two independent sets of requests that each satisfy Definition 3.2-1, and therefore constitute two separate caseation constructs. For good measure, include several pairs of requests that each satisfy the definition of binary alternation.

This program deck can be thoroughly shuffled, compiled, and executed, and the "semantic effect" of the caseations and alternations will be preserved. The execution of the model will "find" those effects; that is its nature.

In the restricted core model caseation appears as an "alternate set", in which the result lists of all requests in the set are identical. From that restriction it would not diminish the utility of the model to add the further restriction that the "alternate set" must be the only requests in an action detail. At this point, neither the mechanical generation of the default conditional nor the verification of the complete disjunction of the existing conditionals is yet practical in the general case. Both, however, are conceptually much more palatable than in the unrestricted model.

3.5 A Proposed Extension: the Concurrent Alternative Construct

There is an important class of problems that deserves special attention in a concurrent model of computation. Consider a case where there is a real choice of algorithms to solve a problem. For a given set of input values, any one of these alternatives might be faster than the others, or slower, or might be incapable of producing an answer. Or there might be a problem in which

one algorithm is guaranteed to give the answer if one exists, but is computationally expensive, while some alternate algorithms which are faster have a finite probability of failure.

An ideal attack in a parallel processing environment would be to start all of these alternative algorithms concurrently, and ensure that as soon as one produces a result the others are terminated. It is not unreasonable to ask that the model make this easy to do.

As a first attempt, consider:

$$\begin{aligned} \text{alg1}(x, y; z) & \left[\overline{z \delta} \right] \\ \text{alg2}(x, y; z) & \left[\overline{z \delta} \right] \\ \text{alg3}(x, y; z) & \left[\overline{z \delta} \right] \end{aligned}$$

where $\overline{z \delta}$ is a mapping that is TRUE when a value for z exists.

It may seem as though this example violates the principle of single assignment, but it is very close to the expression of alternation or caseation within the model. Its justification is that we will endeavor to guarantee that at most one of the effective actions will complete, and therefore only that one will actually produce a value for z . The example will work well if a solution exists, and if one of the algorithms is capable of finding it. If no algorithm succeeds (there is no solution), the program will reach a hang-up condition. In any real situation a programmer would no doubt prefer to report that fact and go on to something else, so it would be nice to find a way to avoid the hang-up. Consider:

$$\begin{aligned} \text{alg1}(x, y; z1, a) & \left[b \vee c \right] \\ \text{alg2}(x, y; z2, b) & \left[a \vee c \right] \\ \text{alg3}(x, y; z3, c) & \left[a \vee b \right] \\ \text{[a] assign}(z1; z) & \\ \text{[b] assign}(z2; z) & \\ \text{[c] assign}(z3; z) & \end{aligned}$$

where a , b , and c are boolean variables.

Here each algorithm *will* complete, unless stopped by its termination condition, and produce a value for its own unique output variable *and* a boolean value. A TRUE indicates that the

output variable ($z1$, $z2$, or $z3$) has the desired value. FALSE indicates that the corresponding z_n has a value (it must if the effective action completed), but that value is trash. The three "assign" requests are effectively a caseation, which selects the value produced by the first-successful algorithm and attaches it to "z". The external termination conditions attempt to guarantee that the first successful completion will terminate (and fail) any remaining survivors. This requires a little caution, as the model must ensure that only one of a , b , and c will receive the value TRUE. The semantics of the evaluation of termination conditions will have to specify precisely when and whether *e.g.* $[b \vee c]$ will terminate "alg1" given that c is TRUE and b has no value. This would indeed be the case if "alg3" completed while the other two actions were still running. This is better than the first example, but it still exhibits the hang-up problem if all algorithms fail (attach FALSE to their boolean results). That can be fixed:

```
alg1 (x, y; z1, a) [ b ∨ c ]
alg2 (x, y; z2, b) [ a ∨ c ]
alg3 (x, y; z3, c) [ a ∨ b ]
[a] assign (z1; z) [not a]
[b] assign (z2; z) [not b]
[c] assign (z3; z) [not c]
[ not a ∧ not b ∧ not c ] assign (FALSE; r)
[ not a ∧ not b ∧ not c ] assign (0; z)
```

This will do what we want, but the notation is complex. As the number of concurrent algorithms increases, readability is rapidly decreased.

If the FAIL() action and the S() and F() predicates are available, we can produce a more succinct version:

```
A: alg1 (x, y; z) [ S(B) ∨ S(C) ]
B: alg2 (x, y; z) [ S(A) ∨ S(C) ]
C: alg3 (x, y; z) [ S(A) ∨ S(B) ]
[ F(A) ∧ F(B) ∧ F(C) ] FAIL()
```

We are now able to produce a formal definition of the concurrent alternative construct, cast in the shape of the last example. A definition done without the S() and F() predicates is also

possible, but more complex.

Definition: a *concurrent alternative* construct is an action detail which consists solely of a set of n requests (n finite) of the form:

$$L_i : \text{action}_i (m_i; r_i) [t_i]$$

where each termination condition t_i is of the form:

$$\bigcup_{j=1}^{j=n} S(L_j) \text{ except for } j = i$$

and where each result list r_i is identical to the result list of the detailed action.

As to how this construct might look in a high-level language, one might expect the following:

Assume that $alg1, alg2 \dots$ are functions that are applied to three arrays to produce two real values, in unpredictable time and with finite probability of failure.

```
x, y := first of
{
    alg1 (a, b, c);
    alg2 (a, b, c);
    alg3 (a, b, c);
    alg4 (a, b, c);
} on failure <statement>;
```

4. Chapter 4 - The Implementation

This chapter describes the implementation of SMART, a primitive language based on the ACM model, and a runtime support system to effect simulated execution of programs written in that language. The name "SMART" was impossible to avoid in view of the fact that an action request in ACM is the five-tuple (s, m, a, r, t) .

4.1 The SMART Language

The language itself is not a high-level language. Its syntax was kept as close as possible to the natural structure of action requests in the model description. The prototype is intended to be used as a tool to investigate the properties of the model itself, rather than the properties of languages derived from the model. If the ACM model is a machine, then SMART is its machine language.

4.1.1 Implemented and Omitted Features This implementation is a starting point, a first step towards what may someday be a full implementation of the ACM model. Only a subset, and a small one at that, of the model's features is supported by the prototype.

As will be apparent from the following list, in many cases the decision not to implement a given feature was not made because the feature was considered "uninteresting". Rather it was made because it was felt that the implementation of that feature was complicated enough to represent a risk to the timely development of the prototype. In other cases, certainly, it was deemed that the prototype would be useful initially without them; they can be added at leisure.

The following list summarizes the restrictions and omissions in the present implementation:

1. All four longevities (fixed, static, dynamic and fluid) are in place.

2. There is no aggregation. All data objects are atomic.
3. Attributes implemented are: *integer, real, character*.
4. In the designator:
 - *Context* is not implemented. All data objects are identified by the simple *user name*.
 - *Spatial position* is not implemented (effectively, there are no arrays). This is regrettable, but arrays cause complications in both the compiler and in the runtime support.
 - *Sequence number* is implemented for dynamics, but *chronological identity* is not.
5. All actions are atomic; no detailing is allowed. The omission of this feature was the hardest decision of all.
6. *Repetition* is not implemented.
7. *Iteration* is implemented, with the extensions proposed in [Fis87].
8. There is no type checking or type conversion. The runtime system will blithely add integer to real, and attach the resulting value to a result object of any type. This is, of course, meaningless.
9. *External* stimulations are supported. *Internal* stimulations are not.
10. Both external and internal *terminations* are supported.
11. Conditional (boolean) expressions in stimulations and terminations are fully supported, including compound conditionals (e.g.: [(a <= b) & (a > c)]).
12. The *S()* and *F()* predicates defined in Chapter 3 of this paper are included.
13. Partialing is not implemented.
14. Designators of actions may *not* appear in the materials or results lists of an action or action request.

4.1.2 *Syntax* The syntax of SMART is the same as that which has been used in the examples in Chapters 2 and 3. The complete BNF is available in Appendix A. The following prototypical action request should be sufficient for most users of the language.

Label: [S_e] name (M-list ; R-list [T_i]) [T_e] ;

Figure 4.1. The SMART Action Request Syntax

In the above statement, the square brackets "[" and "]" must enclose the boolean expressions used as external stimulation (S_e), internal termination (T_i) and external termination (T_e). The action name is any valid symbol (see Appendix A). The materials list is a list of data object names,

separated by commas. The results list is the same. The materials list and the result list are separated by a semicolon. The semicolon after the statement is a statement terminator, as in C, rather than a statement separator as in Pascal. Whitespace is allowed everywhere except within a symbol. An action request may be optionally labeled. If so, the label must be a valid symbol, followed by a colon, placed before the request.

4.1.3 Informal Semantics This section is an informal description of the implementation's static semantics and of the non-obvious semantics of dynamic objects.

4.1.3.1 Object Declarations Contextual declaration of data objects is not allowed. All data objects that appear in material or result lists must be declared at the beginning of a SMART program, before the first action request. The data declaration begins with the keyword "var", followed by any number of data declarations of the form:

```
longevity type symbol ;
```

Figure 4.2. SMART Data Object Declaration

Where longevity is one of *static*, *dynamic* or *fluid*. If omitted, "dynamic" is the default. "Type" must be specified, and is one of *int*, *real* or *char*. "Symbol" is any valid symbol, and must be unique. Static objects (and only static objects) may be "initialized" at declaration time by following the "symbol" with an equal sign ("=") followed by a constant appropriate to the type of the object. See Appendix A for the allowable representations.

The following is an example of a valid set of declarations:

```
var static int max = 255 ;  
    static char c = 'A' ;  
    fluid int i ;  
    dynamic real x ;
```

4.1.3.2 The Semantics of Dynamic Objects References to dynamic data objects may optionally include an explicit specification of sequence number. Two variants exist: absolute and relative. An absolute sequence number is indicated by: *name..i*, where "name" is the simple user name of the object, the two dots are just that, and "i" is an integer *constant*. A relative sequence number is of the form: *name..+i* or *name..-i*, where "i" is again an integer constant. The use of variables or expressions as a sequence number specification is not allowed.

Recall that when a dynamic object appears in the result list of an action request that successfully completes, a new *incarnation* of that object is created, with a sequence number one higher than the previous current incarnation. Initially, an incarnation with sequence number "0" ("x..0") exists for all dynamic objects. At the start of execution, the value component of such objects has not yet been bound, so such an object will not satisfy the data-drive requirements for enabling of an action request. A SMART program will typically contain one or more "assign" actions, such as "assign (5; x)". The statement shown would be immediately enabled (objects of longevity fixed, such as "5" are always complete and available before the inception of the model), and would complete "x..0" by binding its value component.

The "current" incarnation of a dynamic object is synonymous with "object..+0". If a reference to a dynamic object appears in a material list, the default is the current incarnation ("obj..+0"). The default for a dynamic object in a results list is "obj..+1", except in the special case (which actually is the most common case) that "obj..0" exists and is incomplete.

The rationale behind that last statement is that in the typical program most dynamic objects will appear in the result list of one request. The activation of the program will cause each "obj..0" to exist, but have no value component. The completion of the one request that produces the value for each object will then produce the value component of the object, and the object will remain

in that condition for the duration of the program.

The model allows non-determinacy, however, and so does the implementation. We will work with this example:

Assume a, b available, and "a1", "a2" and "a3" will each produce a different value for "x".

```
a1 (a, b; x);  
a2 (a, b; x);  
a3 (a, b; x);  
  
zz (x, y; z);
```

Figure 4.3. Non-Determinacy and Dynamic Objects

The behavior of this program will (or should) exhibit non-determinacy, based on the probably varying relative order of completion of the three actions "a". The problem at hand is determining which "a" will produce "x..0", which "x..1" and which "x..2". There were two possible rules from which to choose while designing the implementation. The one chosen works as follows.

1. No determination is made at the time the actions are enabled, but all three are eligible for simultaneous execution.
2. One will complete before the others, unpredictably. That one provides the value of "x..0".
3. The next action that completes will cause the creation of the new incarnation, "x..1", and will cause a value to be bound to it. Similarly, the third completion creates and binds "x..2".

The request for action "zz" shown in Fig. 4.3 forces another decision. That request is considered for enabling at the same time as the others. The only incarnation of "x" at that time is "x..0", which has no value, so "zz" remains in the idle state. When any request in the model completes, "zz" will be re-examined. At some one of these times, all of its materials are available and it is enabled. At that instant, the implementation determines which incarnation of "x", and therefore which value, "zz" will use. That incarnation is of course the highest absolute numbered instance

of "x", at that instant. Which one that is depends on when "y" becomes available relative to the completions of the actions "a". This is non-deterministic.

Once the semantics in the previous paragraph were chosen, the presence of "obj..+n" in a material list came to mean an unsatisfiable dependency. The compiler rejects it. Also, the appearance of "obj..-n" in a result list is a violation of single-assignment under any reasonable interpretation. That is also rejected by the compiler.

Absolute sequence numbers present some problems. The designation of "obj..99" in a material list is allowed. The dependency will not be satisfied until some action produces the hundredth incarnation of "obj". Something like "obj..99" in a result list leads to enormous problems. Assuming that only "obj..0" exists at the time, what happens when the action with "obj..99" in its result list completes? Specifically, what at that instant is the meaning of "obj..-3"? The compiler rejects any absolute sequence number in a result list; often the best answer to an embarrassing question is to not allow anyone to ask it.

Another semantic problem is the appearance of an object with no value component in a stimulation or termination. Consider:

$$[a < 0] \text{ act } (c, d; e) [(b > 0) | (x = 5)];$$

What should be the effect of this at an instant when neither "a" nor "b" has a value? The implementation provides the intuitively acceptable semantics by declaring $\langle \text{exp} \rangle < \text{relop} \rangle < \text{exp} \rangle$ to be evaluatable, but FALSE, if either or both expressions reference an incomplete object. In the case of the termination condition in the example, if "x" exists and has the value "5", the request *would* be disabled, as the compound conditional becomes "FALSE | TRUE", which is TRUE ("|" is the inclusive or operator, by the way). As with

materials, when dynamic objects with multiple instances are used in boolean expressions, the value of the incarnation with the highest sequence number *at the instant of evaluation* is used.

4.2 Implementation Details

The implementation is running currently on five AT&T 3B2/300 processors, connected by ETHERNET, and using the UNIX⁹ System V Release 2 operating system.

4.2.1 Compiler A "program" written in the SMART language must be first submitted to a compiler, the internals of which are described in [Shi87] and will not be treated here. It is sufficient to state that the compiler produces an intermediate form of the program in a UNIX system file. This file is essentially a flat representation of the dataflow graph of the program, which is requested and read at "run time", by a "virtual machine loader", and finally executed.

4.2.2 Node Modules An entity, called a "node", is started manually on a host machine by typing a single command ("smart", naturally). This invokes a process called the Node Manager, who in turn uses the UNIX system *fork* and *exec* commands to create the other three processes that constitute the basic idle node. These processes are the Terminal Driver, the Scheduler, and Network Driver. Other processes, incarnations of an entity known as the Environment Handler, are created (by the Scheduler) as needed to effect the execution of one or more SMART programs.

This apparently wanton proliferation of processes is necessary to achieve the desired maximally independent execution of actions, combined with atomicity in the effects of the completion of

9. UNIX is a registered trademark of AT&T.

such actions on other requests. There is surprisingly little use of signals; most communication between the processes is done with UNIX system message queues. Indeed, signals are used only to effect a manually requested orderly shutdown of the node.

4.2.2.1 Node Manager The Node Manager, once it has created the other node processes, then enters a mode in which it receives messages placed on its own private queue and analyzes them to determine which service is being requested. It then either performs the service itself, such as logging and node shutdown, or routes the message to some other process for action.

4.2.2.2 Terminal Driver The primary function of the Terminal Driver is to wait for input from the terminal and forward it to the Node Manager. The Terminal Driver also detects the "break" key, asks for a "we really mean it" from the operator, and instructs the Node Manager to shut down the node.

4.2.2.3 Scheduler When a request comes in from the terminal to begin execution of a SMART program file (the output of a compilation), the Scheduler creates a new Environment Handler to read and interpret the dataflow graph produced during compilation. There may be several Environment Handlers (programs) active on the node at any time. When an Environment Handler enables any action request, it sends a message to the Scheduler, who creates a task to execute the action. The actions in this implementation are really executable UNIX files in an "action library". This is, of course, in lieu of true detailing. The Scheduler forwards a data structure to the task, gets it back when the task terminates, and passes it back to the Environment Handler. This structure contains an action designator, the values of the objects in the action's material list, and space to carry the values to be attached to the action's result objects. The structure bears a close, entirely unaccidental, resemblance to Dennis's *actors* [Den80]. The Scheduler detects catastrophic failure of a task, and returns the "actor" to the

Environment Handler with no results and with an indication of the failure of the action.

The Scheduler is also in communication with its counterparts on other nodes. When an Environment Handler requests execution of an action, the Scheduler makes a random choice of the node to do it. If the decision is to do it on the local node, the Scheduler creates and monitors a task as described above. Otherwise the Scheduler sends the "actor" over the network to the Scheduler on the selected remote node. Meanwhile, the Schedulers on the other nodes are doing the same thing (the nodes are all peers). When such a request from another node arrives, the Scheduler creates a task, monitors it, and returns the results to the Scheduler on the remote node.

4.2.2.4 Environment Handler The Environment Handler is essentially an interpreter of the dataflow graph built by the compiler. This graph is implemented as a set of interconnected data structures built and linked by the compiler. There is an "object block" for each object, which contains the information that embodies the model's concept of an object (or at least the implemented subset). The special object known as an action has its own control block, the "action object block", which contains such things as pointers to "object lists". An object list is an array of pointers to the object blocks that represent the objects in an action's material or result list.

The Environment Handler's operation is conceptually simple, but the actual code looks complicated because of the intensive pointer manipulation involved in relating the many control blocks. Initially all objects, except those of longevity fixed and compile-time initialized statics, are unavailable (have no value). All actions are in the idle state, since in the absence of detailing the entire program is a single large request set.

The list of idle actions is then searched to find those actions whose data drive and stimulation

requirements are satisfied. At this point, this can only result from the presence of fixed or initialized static objects. All such requests are immediately enabled, and a request for the execution of each is sent to the Scheduler.

The "actor" structure mentioned above in the description of the Scheduler is not really a structure at this time. Because the execution of an action is effected by the use of the UNIX system's *exec* call, the material values passed to the execution task must be in the form of ASCII strings. The Environment Handler must build these from the values in the material object blocks.

When an action terminates, the Scheduler returns an item that looks more like an "actor", and which contains the values to be associated with the objects in the action's result list. If the action failed, this is indicated and the result objects are not updated. The action block is disabled in either case.

As each action terminates, all idled actions remaining must be examined to determine whether the results of the just-terminated action have made any of them eligible to be enabled. At the same time, enabled actions with termination conditions must also be examined. The required atomicity of the effects of completion of an action is achieved here. All changes of state of objects and other actions are done in one entry to the Environment Handler. The "concurrent" termination of another action cannot interfere, as there is only one Environment Handler for each SMART program.

At the present time, the Environment Handler does a linear search over a list of action blocks. It would have been possible, but difficult in the compiler, to have each object point to all actions in whose material list or stimulations and terminations that object appeared. This may or may not be an area for future performance enhancement. A typical SMART program will probably not be large, and will probably have more data objects than action objects. That would tend to

negate any performance gain from such a change. It is also not clear that performance is an important issue anyway.

A complication arises when multiple incarnations of a dynamic object exist. Such an entity has a single object block, and values of previous incarnations are maintained in an auxiliary value block chained to the basic object block. The number of previous incarnations kept is open-ended. New auxiliary blocks are dynamically acquired and chained as needed.

Iteration causes several complications. The "local context" of any iteration is destroyed when the internal termination condition is checked, and only the final values of those objects impact the outside world. This is done by using a duplicate set of object lists for materials and results within the local incarnation. Although it is required that the internal termination condition of an iteration be a function of a dynamic variable for which a new incarnation is created within the loop, these expressions might well involve "outside" objects as well. The value of the incarnation of such an object that was current when the iteration started must therefore be "frozen" in the action block for this use.

4.2.2.5 Network Interface The Network Handler process is created by the Node Manager when the node is initialized. In the present implementation, it is the Network Handler who prompts the operator for the name of the node. To the current ETHERNET driver within the UNIX system, a logical node address is a mostly arbitrary six-byte field. The Network Handlers on all nodes use the same bits in the first five bytes of this address, and append the eight-bit binary value of their own node numbers as the sixth byte. In this way, they can easily generate the ETHERNET address of any other node.

The Network Handler establishes an ETHERNET duplex virtual port by means of a few simple UNIX system calls, and then does a UNIX *fork*, producing an exact copy of itself. This "child"

process inherits such things as open files from its parent, and access to the ETHERNET is essentially one of those. The child process does all network input for the node, while the parent process does all the node's output.

This is a good example of the reasoning behind the many-process design of the implementation. Most of the time, the net input process is waiting for incoming traffic from the bus. When it comes, the process immediately awakens and passes the incoming message to the Scheduler. The net output process, on the other hand, is constantly waiting for a message to appear on its message queue. When one comes, the process immediately sends it out to the net. The code is extremely simple done this way; done with one process, it would be much more complex.

When the node first comes up, the network output process sends a message to all possible SMART nodes. Any nodes that are already up will receive this message and will respond with their own node numbers. The Scheduler on each node sees these messages and thus knows which nodes are active. This information is of course used when sending actions for execution on other nodes.

When a node is taken down the last act of the Network Handler is to broadcast a farewell message, which is also seen by the remaining Schedulers. Any Scheduler that has tasks still active on the departing node will immediately reschedule them on a surviving node. This works if the node is taken down in an orderly manner (by operator request), but not if a node fails by virtue of a software fault or a failure of its host machine. It is possible, but not really worth the effort, to add code to send periodic sanity checks from node to node to detect this condition.

4.2.2.6 Atomic Action Library As mentioned earlier, atomic actions in the implementation exist as executable files that are invoked by means of the UNIX system *exec* call. This is, of course, grossly inefficient in terms of execution time, but it adds great flexibility from the standpoint of

adding new actions, and it contributes very nicely to the indeterminacy of order of completion of concurrent actions.

There is one unusual set of actions contained in the library. Each does one simple numeric calculation by the simple expedient of sending a question to the console of whatever node it is running on. One might see, for example,

12: What's 15 + 25?

The "12:" in the above example is an identifier added by the Node Manager, and is used to route the answer back to the requesting task. By choosing the order in which these replies are given, the operator can exercise conscious control over the order of completion of actions to determine whether a given algorithm is sensitive to that order. Note that this technique is just as effective on a single node as on many, and it's a lot more convenient.

5. Chapter 5 - Conclusions

The ACM model is an abstract model of concurrent computation. As such, its function is to provide a mathematically provable semantic base for real products such as programming languages or modeling and simulation languages.

5.1 Results of this Research

When such a model is reviewed, however closely, in the abstract, it is inevitable that some omissions will evade detection. It requires the hard practical scrutiny of many people, actually trying to solve problems by manipulating the model, to provide the feedback for the ultimate polishing of such a model. The purpose of the prototype implementation described by this paper has been the production of a tool to aid that investigation.

The implementation of such a prototype is itself a practical experience with the model. That experience has already resulted in the elimination of some ambiguities from the original model, and the suggestion of several additions, reported in this and two related papers ([Fis87] and [Shi87]).

The SMART language itself is not seriously suggested as a vehicle for performing practical computations. It is intended to be used as a tool for hands-on manipulation of the model itself. Consider it, if you will, the "assembly language" of the model.

The corrections and additions proposed in Chapter 3 of this paper include:

- A tightening of the definition of states and state changes of an effective request, including a formal distinction between "failure" and "[successful] completion" of a request or request set.

- The definition of built-in predicates to make available the knowledge of completion (successful or failure) of a request, for more succinct expression of conditional computation based on such completion. Also included is the idea of a mechanism whereby a request set can intentionally effect its own failure, with all that that failure implies.
- An addition to the definition of caseation to account for the possibility that during execution *no* alternative might be selected. The suggestion that the notion of *default case* be legitimized by the model is strictly due to the personal bias of the author. It is by no means unanimous among language experts that such a facility is desirable.
- The definition of the *concurrent alternative* construct, a formalism which would seem to be useful in (and unique to) concurrent languages.

5.2 Future Directions

The prototype as it now exists implements a small subset of the ACM model. It works, and it has just enough power for other investigators to start gaining experience with the model.

By far the most conspicuous by its absence is the entire concept of action detailing. This was omitted with the greatest reluctance, for its absence precludes the investigation of the most interesting features of the model. There were firm time constraints on the development of the prototype, and it was feared that an attempt to implement detailing would jeopardize the availability of the entire product. The addition of detailing to the prototype is urgently suggested as a very near future project.

Another noticeably absent feature is the spatial position subcomponent of instance. In other words, arrays. The added complications that arrays cause in a compiler and in a run-time support system make their omission from a time-constrained prototyping effort most attractive,

and this case was no exception. Unfortunately, arrays are necessary for the investigation of many interesting problems. The addition of spatial position is encouraged.

The above-mentioned omission of spatial position from the prototype precluded the implementation of repetition. Iteration appeared to be the more interesting case in an implicitly concurrent single assignment model. Repetition is at least theoretically well-understood. However, the practical problem of the automatic recognition, by a high-level language, of the opportunity for concurrent execution of repetitive requests is truly formidable. Repetition should be introduced into the prototype as soon as arrays are.

As mentioned in Chapter 4, the implementation is a little fragile in the face of a catastrophic failure of a node. It would be possible to recover from that by, for example, the periodic exchange of a "sanity" message between nodes. This problem is just a minor annoyance, but its repair could provide a pleasant hour's diversion for someone.

It would also be convenient to run multiple nodes on a single processor, which is not easily done now. The processes that use message queues use hard-compiled keys to establish them. An experienced UNIX programmer could easily change that code to generate a unique key for each node (from a command line parameter for example) and pass that key, or an actual queue identifier, to each process as part of the *exec* argument list. There will no doubt be dozens of similar projects available.

The entire concept of input-output and files was avoided in (evaded by?) the prototype, but this implementation is certainly not an historical first in that regard. Files can be made to work, but there is room for a great deal of philosophical speculation in the area of *sequential* files. Does the notion of sequential file even have a right to exist in a concurrent world, or is it just another fiction forced upon us by the limitations of sequential computers? Surely the alphabetically

arranged dictionary existed on paper long before computers. But just as surely, the mechanism whereby the human mind *recalls* a word must be much closer to the operation of a set-associative memory than to a sequential search. It is only when looking up a word in a paper dictionary that we are reduced to a variant of a binary search on an ordered set of data. Is this anything more than a limitation of our tools?

When faced with questions like that, this author feels an uncomfortable kinship with the fifteenth century mariner, who sees at the edge of his chart the legend: "Beyond here be there dragons". With a reluctance not untainted by a certain guilty relief, we both leave it to better equipped and more intrepid adventurers to go chart those waters.

Bibliography

- [Ack79] Ackerman, W.B., Data Flow Languages, Proceedings of the National Computer Conference, Volume 48, 1979, pages 1087-1095.
- [Alm85] Almasi, G.S., Overview of Parallel Processing, *Parallel Computing* 2 (1985), pgs 191-203.
- [And83] Andrews, G.R., and Schneider, F.B., Concepts and Notations for Concurrent Programming, *Computing Surveys*, Vol 15, No 1, March 1983, pgs 3-43.
- [Arv80] Arvind, Kathail, V., and Pingali, K., A Data Flow Architecture with Tagged Tokens, Laboratory for Computer Science, Technical Memo 174,
- [Bac78] Backus, J., Can Programming Be Liberated from the von-Neumann Style? A Functional Style and Its Algebra of Programs, *Communications of the ACM*, Vol. 21, No. 8, August 1978
- [Bra84] Bracchi, G. and Pernici, B., The Design Requirements of Office Systems, *ACM Transactions on Office Information Systems*, Volume 2, No 2, April 1984, Pages 151-170.
- [Bru83] Bruner, J.D., and Reeves, A.P., A Parallel P-Code for Parallel Pascal and Other High Level Languages, Proceedings of the 1983 International Conference on Parallel Processing, pages 240-243.
- [Com78] Comte, D. et. al., Parallelism, Control, and Synchronization Expression in a Single Assignment Language, *SIGPLAN Notices*, January 1978.
- [Cyt82] Cytron, R.G., Improved Compilation Methods for Multiprocessors, U of I Report No. UTUCDCS-R-82-1088, UILU-ENG 82 1713, May 1982.
- [Dav81] Davies, J.R.B., Parallel Loop Constructs for Multiprocessors, U of I Report No. UTUCDCS-R-81-1070, UILU-ENG 81 1719, May 1981.
- [Dav82] Davis, A.L., and Keller, R.M., Data Flow Program Graphs, *IEEE Computer*, February 1982, pgs 26-41.
- [Den80] Dennis, J.B., Dataflow Supercomputer, *IEEE Computer*, 13(11) (1980), pgs 48-56.
- [Den83] Dennis, J.B., and Rong, G.G., Maximum Pipelining of Array Operations on Static Data Flow Machine, Proceedings of the 1983 International Conference on Parallel Processing, pages 331-334.
- [DeR76] DeRemer, F., and Kron, H.H., Programming-in-the-Large Versus Programming-in-the-Small, *IEEE Transactions on Software Engineering*, Vol SE-2, No. 2, June 1976, pgs 80-86.
- [Diet85] Dietz, H., and Klappholz, D., Refined C: A Sequential Language for Parallel Programming, Proceedings of the 1985 International Conference on Parallel Processing, pgs 442-449.

- [Dij68] Dijkstra, E.W., Cooperating Sequential Processes, In Programming Languages, F.Genuys, Ed., Academic Press, New York, 1968, pgs 43-112.
- [Ens74] Enslow, P.H. (editor), Multiprocessors and Parallel Processing, John Wiley & Sons, Inc., 1974.
- [Ezz85] Ezzat, A.K., and Agrawal, R., Making Oneself Known in a Distributed World, Proceedings of the 1985 International Conference on Parallel Processing, pgs 139-142.
- [Fis87] Fish, R. W., Extensions to the ACM Dataflow Model, Masters Thesis, Kansas State University, 1987 (in preparation)
- [Fly79] Flynn, M.J., Computer Organizations and Architecture, in Operating Systems An Advanced Course, ed. Bayer, R. et. al., Springer-Verlag, 1979.
- [Frab85] Fraboul, C., and Hifdi, N., Expressing and Exploiting Parallelism on an Experimental MIMD System, Proceedings of the 1985 International Conference on Parallel Processing, pgs 236-238.
- [Fran85] Francez, N., and Yemini, S.A., Symmetric Intertask Communication, ACM Transactions on Programming Languages and Systems, Vol 7, No 4, October 1985, Pages 622-636.
- [Feld79] Feldman, J.A., High Level Programming for Distributed Computing, Communications of the ACM, Volume 22, Number 6, June 1979, pgs 353-368.
- [Gaj82] Gajski, D.D., Padua, D.A., Kuck, D.J., and Kuhn, R.H., A Second Opinion on Data Flow Machines and Languages, IEEE Computer, February 1982, pgs 58-69.
- [Gaj83] Gajski, D., Kuck, D., Lawrie, D., and Sameh, A., Cedar - A Large Scale Multiprocessor, Proceedings of the 1983 International Conference on Parallel Processing, pages 524-529.
- [Gaj85] Gajski, D.D., and Peir, J.K., Comparison of five multiprocessor systems, Parallel Computing 2 (1985) pgs 265-282.
- [Gan75] Gannon, J.D., and Horning, J.J., Language Design for Programming Reliability, IEEE Transactions on Software Engineering, Vol SE-1, No. 2, June 1975, pgs 179-191.
- [Geh84] Gehani, N., Ada Concurrent Programming, Prentice-Hall, 1984.
- [Gel76] Gelly, O. et. al., LAU System Software: A High Level Data Driven Language for Parallel Programming, Proceedings of the 1976 International Conference on Parallel Processing, pg. 255.
- [Ghe85] Ghezzi, C. Concurrency in Programming Languages: A Survey, Parallel Computing 2 (1985) pgs 229-241.
- [Gur85] Gurd, J.R., Kirkham, C.C. and Watson, I., The Manchester Prototype Dataflow Computer, CACM January 1985, Volume 28, Number 1.
- [Hal85] Halstead, R.H., Multilisp: A Language for Concurrent Symbolic Computation, ACM Transactions on Programming Languages and Systems, Vol 7, No 4, October 1985, pages 501-538.

- [Han73] Hansen, P.B., Operating System Principles, Prentice-Hall, Inc., 1973.
- [Han75] Hansen, P.B., The Programming Language Concurrent Pascal, IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pages 199-207.
- [Har85] Harris, J.P., Representing and Effecting Parallelism in Programs for Control-Flow Multiprocessors: Program Graphs and a Program Graph Interpreter, U of I Report No. UIUCDCS-R-85-1200, UIIU-ENG-85-1705, January 1985.
- [Her84] Herlihy, M.P., Replication Methods for Abstract Data Types, PH.D. Dissertation, June 1984, MIT.
- [Hib78] Hibbard, P., Hisgen, A., and Rodeheffer, T., A Language Implementation Design for a Multiprocessor Computer System, Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978, pages 356-362.
- [Ho83] Ho, L.Y., and Irani, K.B., An Algorithm For Processor Allocation In A Dataflow Multiprocessing Environment, Proceedings of the 1983 International Conference on Parallel Processing, pages 338-340.
- [Hoa72] Hoare, C.A.R., Towards a theory of parallel programming, in Operating Systems Techniques, Academic Press, New York, 1972, pgs 61-71.
- [Hoa74] Hoare, C.A.R., Monitors: An operating system structuring concept, Communications of the ACM, Volume 17 Number 10, 1974, pgs. 549-557.
- [Hoa78] Hoare, C.A.R., Communicating Sequential Processes, CACM August 1978, Volume 21, Number 8, pages 666-677.
- [Jon80] Jones, A.K., and Schwarz, P., Experience Using Multiprocessor Systems - A Status Report, ACM Computing Surveys, Volume 12, No 2, June 1980, pages 121-165.
- [Kel85] Keller, R. M. and Lindstrom, G., An Approach to Distributed Database Implementations Through Functional Programming Concepts, Proceedings of the International Conference on Distributed Computing Systems, May 1985
- [Kuc74] Kuck, D.J. et. al., Measurements of Parallelism in Ordinary FORTRAN Programs, Computer, Volume 7, Number 1, January 1974.
- [Kog85] Kogge, P.M., Function-based computing and parallelism: A review, Parallel Computing 2 (1985), pgs 243-253.
- [Lel83] Leler, W., A Small, High-Speed Dataflow Processor, Proceedings of the 1983 International Conference on Parallel Processing, pages 341-343.
- [Lie85] Liebowitz, B.H., and Carson, J.H., Multiple Processor Systems for Real-Time Applications, Prentice Hall, 1985.
- [Lis77] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction Mechanisms in CLU, Communications of the ACM, Volume 20, Number 8, August 1977, pgs 564-576.
- [Mot85] Motteler, H.E., and Smith, C.H., A Complexity Measure for Data Flow Models, International Journal of Computer and Information Sciences, Vol 14, No 2, 1985, pgs 107-122.

- [McG82] McGraw, J.R., The VAL Language: Description and Analysis, ACM Transactions on Programming Languages and Systems, Vol 4, No 1, January 1982, pgs 44-82.
- [Pad80] Padua, D.A., Kuck, D.J., and Lawrie, D.H., High Speed Multiprocessors and Compilation Techniques, IEEE Transactions on Computers, Vol. C-29, No. 9, Sept. 1980, pgs. 763-776.
- [Par72] Parnas, D.L., On the Criteria to be used for Decomposing Systems into Modules, Communications of the ACM, Volume 15, Number 12, 1972, pgs 1053-1058.
- [Per79] Perrott, R.H., A Language for Array and Vector Processors, ACM Transactions on Programming Languages and Systems, Volume 1, Number 2, October 1979, pages 177-195.
- [Pet77] Peterson, J.L., Petri Nets, Computing Surveys, Vol 9, No 3, September 1977, pgs 223-252.
- [Pfi85] Pfister, G. F., et. al., The IBM Research Parallel Processor Prototype (RP3), Introduction and Architecture, IBM T. J. Watson Research Center, 1985
- [Rei79] Reif, J.H., Data Flow Analysis of Communicating Processes, Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages, January 1979, pgs 257-268.
- [Sch83] Schwartz, J., A Taxonomic Table of Parallel Computers, Based on 55 Designs, Ultracomputer Note #69, Courant Institute, New York University, 1983
- [Shi87] Shirley, T. E., A Programming Language and Compiler Based on an Augmentation of the ACM Model, Masters Thesis, Kansas State University, 1987 (in preparation)
- [Sri85] Srinivasan, V.P., An Architecture for Doing Concurrent Systems Research, AFIPS Conference Proceedings, Volume 54, 1985 National Computer Conference, pages 267-277.
- [Ung78a] Unger, E.A., A Natural Model for Concurrent Computation, Kansas State University Technical Report 78-35, Dissertation 1978.
- [Ung78b] Unger, E.A., and Schweppe, E.J., A Concurrent Model: Basic Concepts, European Conference in Parallel and Distributed Processing, 1978.
- [Wei71] Weinberg, G.M., The Psychology of Computer Programming, New York: Van Nostrand Reinhold, 1971.
- [Wil84] Williamson, R., and Horowitz, E., Concurrent Communication and Synchronization Mechanisms, Software Practice and Experience, February 1984, Volume 14(2), pages 135-151.
- [Woo83] Woo, N.S., and Agrawala, A.A., The DC1 Flow Schema with the Data/Control-driven Evaluation, Proceedings of the 1983 International Conference on Parallel Processing, pages 244-251.
- [Wul76] Wulf, W.A., London, R.L., and Shaw, M., Abstraction and Verification in Alphard, IEEE Transactions on Software Engineering, April 1976.
- [Zar85] Zargham, M.R., and Purcell, R.D., A Protocol for Load Balancing on CSMA Networks, Proceedings of the 1985 International Conference on Parallel Processing,

pgs 163-165.

Appendix A - SMART BNF

```
pgm      :   varlist reqlist

varlist  :   VAR dclist
           |   /* Null */

dclist   :   dclist dcitem ;
           |   dcitem ;

dcitem   :   longevity stype ID
           |   longevity stype ID ( INTVAL ) /* not implemented */
           |   longevity stype ID = sign INTVAL
           |   longevity stype ID = REALVAL
           |   longevity stype ID = sign CHARVAL

longevity:  FLUID
           |  DYNAMIC
           |  STATIC
           |  FIXED
           |  /* Null */

stype    :   INT
           |   CHAR
           |   REAL
           |   FILE           /* not implemented */

reqlist  :   reqlist request ;
           |   request ;

request  :   label se ID rspec te
           |   se ID rspec te

label    :   ID ;

rspec    :   ( mlist ; rlist ti )

mlist    :   mlist , desig
           |   /* Null */

rlist    :   rlist , desig
           |   desig
           |   /* Null */
```

```
se      :      [ cond ]
         |      /* Null */

te      :      [ cond ]
         |      /* Null */

ti      :      [ cond ]
         |      /* Null */

sign    :      +
         |      -
         |      /* Null */

desig   :      username instance

username :      ID

instance :      .. sequence
         |      spatialpos . sequence /* not implemented */
         |      .. spatialpos      /* not implemented */
         |      /* Null */

spatialpos:      ( desig ) /* not implemented */
         |      ( INTVAL ) /* not implemented */

sequence:      INTVAL
         |      + INTVAL
         |      - INTVAL

cond     :      x bprime

bprime  :      OR x bprime
         |      /* Null */

x       :      y xprime

xprime  :      AND y xprime
         |      /* Null */

y       :      ( cond )
         |      boolexp
         |      S ( ID )
         |      F ( ID )

boolexp :      expr relop expr

expr    :      t eprime
```

```
eprime :   + t eprime  
         |   - t eprime  
         |   /* Null */  
  
t       :   f tprime  
  
tprime :   * f tprime  
         |   / f tprime  
         |   /* Null */  
  
f       :   ( expr )  
         |   desig  
         |   sign INTVAL  
         |   sign REALVAL  
  
relop  :   EQ  
         |   NE  
         |   GE  
         |   GT  
         |   LT  
         |   LE
```

Appendix B - SMART User Notes

The following is an overview of how to use SMART:

Compilation

In order to compile a program enter "acm filename" where filename is the name of the source file. The compiled output will be called "filename.out" unless the option "-o filename" is entered on the compilation command, in which case the name of the compiled output will be "filename".

Starting or stopping a node

To start a node simply enter "smart". You will be prompted for a unique node name. If any other nodes are up or come up they will be automatically attached without manual intervention. In order to terminate a node simply enter "exit". Any tasks being performed on that node for another will restart elsewhere.

Answering Questions

Any questions displayed on the terminal are prefixed with "nn: ". In order to reply the user should attach the same prefix to the reply. Entering "?<cr>" will redisplay any outstanding questions.

Starting a Program

Simply enter the name of the compiler output file

Debugging a Program

The following commands may be entered in order to debug a program or a node control problem:

- list jobs - lists all active programs
- list tasks- lists all active tasks
- list all - does both of above
- list active - displays start/stop of tasks
- list silent - logs start/stop of tasks
- list stop - stops a active/silent display
- log "xxxx" - logs a comment
- dump "nnn" - dumps data flow graph status
- kill "nnn" - terminates an active program
- "@" - line kill character
- "!" - shell escape

Additionally a disassembler is available which will produce a listing of requests with identification numbers which will associate to the various display commands above. This

disassembler is invoked, outside of the node, by entering "acmdis filename".

Debugging Extensions

If extensions are made to the language, the capability to interactively debug the environment handler can be done using two terminals. The node must be started on one terminal. On the other terminal, using "sdb", the environment handler is started. Determine the process number and on the other terminal (running smart) enter "debug nnn" where "nnn" is the environment handler process number. At this point the environment handler will be able to have tasks executed.

SMART: A Tool for the Study of the ACM Model of Concurrent Computation

by

Richard Edward Yuknavech

B. S., St. John's University, 1960

AN ABSTRACT OF A MASTERS THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

Parallel processors are currently perceived as the most likely source of badly needed increased processor speed. Efforts in designing programming languages for such processors have been hampered by the fact that these represent a radical departure from the traditional von Neumann architecture upon which most of our perspectives are based.

"A Concurrent Model" (ACM) is an abstract model of computation that combines intrinsic concurrency, data drive and optional explicit control specification in a single tool that has the power to naturally describe concurrent computation over a broad spectrum of environments.

This paper describes a working implementation of a tool to manipulate programs written in the "machine language" of the ACM model. It is intended to be used as an instructional and research tool, providing "hands on" experience with the ACM model itself. Corrections and extensions to the model which arose during the implementation process are discussed.