

Extensions to the ACM Dataflow Model

by

Robert Wayne Fish

B. S., University of Denver, 1969

A MASTERS THESIS

submitted in partial fulfillment of the

requirements for the degree

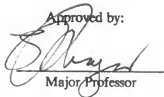
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:



Major Professor

0
668
74
MSC
987
-57
L-2

A11207 308230

Acknowledgements

I would like to thank T. E. Shirley and R. E. Yuknavech for the many hours of discussion about the ACM model and the ramifications of the various semantic problems encountered within the implementation. Of course Dr. E. A. Unger must be thanked as both the author of the ACM model as well as being our guide through the pitfalls of misinterpretation of it.

I would particularly like to thank my wife, Carol, for putting up with me for the last year.

CONTENTS

1. The Problem	1
1.1 Introduction	1
1.2 Relevant Ideas from Current Research	2
1.3 The Goal of This Paper	5
1.4 Summary of Contents	6
2. The ACM model	8
2.1 Overview of the Model	8
2.2 Objects	8
2.3 Actions and Requests	10
2.4 Control Constructs	12
2.5 Summary	15
3. Analysis and Redefinition	16
3.1 Introduction	16
3.2 An Extension to the Iteration Construct	16
3.3 The Deferred Execution Construct	22
3.4 Operations on Unordered Collections	25
3.5 Mapping Coercion in Repetition Constructs	26
3.6 A Summary of Other Research	28
4. SMART - an ACM Language (the Semantics)	30
4.1 Introduction	30
4.2 Limitations on the Implementation	30
4.3 Informal Semantics	31
5. SMART - an ACM Language (The Implementation)	38
5.1 Introduction	38
5.2 Compilation	38
5.3 The SMART Network	38
5.4 The SMART Node	39
5.5 The SMART Library	42
5.6 User Instructions	43
6. Conclusions	45
6.1 Accomplishments	45
6.2 Notable Limitations	46
6.3 Areas for Future Research	47
The SMART Node: User Notes	49
The SMART Library	51
The SMART BNF Syntax	52
Smart Sample Program 1	55
SMART Sample Program 2	58
Bibliography	59

LIST OF FIGURES

Figure 2-1. Sample Action Declaration	11
Figure 2-2. Sample Request Declaration	11
Figure 2-3. Detailing	12
Figure 2-4. Locality of Context Within an Action Detail	12
Figure 2-5. Alternation	13
Figure 2-6. Caseation	13
Figure 2-7. Repetition	14
Figure 2-8. Iteration	14
Figure 2-9. Partialing	15
Figure 3-1. Iteration Problem 1	17
Figure 3-2. The Material and Results of Iteration Problem 1	17
Figure 3-3. Iteration Problem 2	18
Figure 3-4. The Material and Results of Iteration Problem 2	18
Figure 3-5. Iteration Problem 3	19
Figure 3-6. The Material and Results for Iteration Problem 3	20
Figure 3-7. Deferred Execution as Currently in the Model	23
Figure 3-8. Potential Deferred Execution Construct	24
Figure 4-1. Data Object Declarations	31
Figure 4-2. Data Object Value Initialization	32
Figure 4-3. A General Request Declaration in SMART	32
Figure 4-4. A Simple Request Declaration in SMART	33
Figure 4-5. Correct Use of Dynamic Data Objects	33
Figure 4-6. Incorrect Use of Dynamic Data Objects	34
Figure 4-7. Iteration	34
Figure 4-8. Internal Versus External Termination	35
Figure 4-9. Success or Fail within Conditions	36
Figure 4-10. Condition Evaluation	36
Figure 5-1. The SMART Network	39

Figure 5-2. The SMART Node 40

Chapter 1

1. The Problem

1.1 Introduction

Traditional computers have been based upon the Von-Neumann style of architecture which consists of one processor, one memory space, and sequential, one at a time, execution of instructions. Advances in technology have allowed significant increase in performance of computers within the constraint of this design, however further increases are limited due to the physical laws of nature. The speed of a circuit is a function of its physical length and the minimal length of circuits is rapidly being reached.

Advances in VLSI technology have greatly reduced the cost of both processors and the memory accessed by them; the financial constraint of hardware has been broken. Computers can now be economically constructed of multiple processing elements with large amounts of memory. It would appear that the only significant means to gain speed is through the exploitation of the potential overlapping of operations possible through simultaneous use of multiple processors. Various techniques such as vectoring and pipelining allow significant performance enhancement through the simultaneous execution of multiple instructions but even this is not enough if the programming languages remain essentially sequential.

Most languages in wide use today were developed within the constraints of Von-Neumann architecture and were designed to execute in a sequential manner. While these are sufficient for such machines, the advent of multi-processor complexes demands that they be exploited and thus the trend seems to be towards the introduction of language primitives which allow for concurrency of operation.

The introduction of concurrency within imperative languages introduces a new set of difficulties which arise primarily from the sequential origins of such languages. As an example, the access permission to globally shared data by concurrently executing tasks can cause loss of control if such permissions include the ability to update. The burden placed upon the programmer to correctly control such data may be extreme and thus cause more effort to be placed on how a program works rather than upon what it accomplishes [ALM85].

As stated by [GAN75]:

"Programming Languages should lead to confidence in the correctness of programs...language should contain features that allow violations of a programmer's intentions to be detected as errors."

If such a problem as stated above exists in the design of a language then neither of these goals have been met; much of the current research in concurrent programming have them as an underlying premise. The means to achieve them may differ dramatically; one language may provide a rich set of synchronization primitives with a rigid definition for their use, while another may attempt to shield the programmer from any knowledge of what concurrency will occur.

1.2 Relevant Ideas from Current Research

Most existing concurrent programming languages are essentially sequentially imperative languages with concurrency primitives built in; the ordering of the statements is critical. Control of concurrency is achieved through various synchronization primitives. As an example Concurrent Pascal [HAN75] uses monitors as the means to update global data areas; this serializes their use. Various other structures such as queues constructed using monitors are used as the means for ordering activities between the concurrent processes of a program.

CSP (Communicating Sequential Processes) as proposed by [HOA78] makes use of input and output primitives as the basic communication between parallel sequential processes. Guarded commands [DIJ75] are employed as the only sequential control structure; the successful execution of a guard "stimulates" the guarded instructions to be executed. Failure of a guard will result in the guarded command not being executed.

Object oriented languages such as CLU [LIS77] allow the programmer to think in terms of the objects to be manipulated. Data abstraction provides a mechanism to define the objects into sets of data objects, which may be atomic or aggregate, which carry with them information which establishes the essence of what the objects represent. The language CLU implements this concept through the use of "clusters" which couple with the objects the set of operations permissible on them.

Explicit declaration of parallel arrays in ACTUS [PER79] allows the programmer to exploit the power of vector processors without otherwise being overtly concerned with concurrency. This is yet another example of data abstraction. A user effectively declares which elements will not interfere with each other during an iterative manipulation and thus which iterations can proceed in parallel.

In the language MULTILISP [HAL85] the creation and synchronization of tasks is controlled using the "future" construct. A "future" is a data abstraction which allows the use of an object with an undetermined value to proceed in parallel with other executing tasks, one of which may be actually computing the value. Once established the value is said to be "determined". Any operation that needs to know the value of a future which is as yet undetermined will suspend, but many operations such as assignment do not need the actual value. As a result a significant amount of parallelism can occur through the declaration of futures. Coupled with futures are

"delay" constructs which act like futures but explicitly delay the determination of the value until such time as the value is required. This allows the language to indefinitely defer performing activity unless that activity is really needed.

The concept of dataflow languages is not new [DEN72] but they have evolved slowly, perhaps due to the tremendous advances in performance heretofore achieved on traditional Von Neumann architecture. Dataflow languages essentially execute dataflow graphs [DAV82] where the order of execution is entirely dependent upon the availability of data. A transition (executable activity) occurs whenever all of its input data items (tokens) are available. Such a transition is said to be "enabled" and its actual execution to being "fired". The flow of execution is controlled entirely by the availability of tokens. As many enabled transitions can fire simultaneously as can be scheduled with their ordering nondeterministic (and really irrelevant). The inherent parallelism is discovered when the dataflow graph is constructed, at compile time, without overt attention being placed on it by the programmer. The programmer is freed to consider the problem at hand rather than upon how it gets accomplished.

The ability to access and modify global data areas, or, in other words, "side effects", is the bane of concurrency. Most, if not all, languages designed for parallel execution make provision to control or eliminate global update. The concept of single assignment [COM76] eliminates side effects by never allowing a value to change once it has been set. New instantiations of data are created whenever a value needs to change. As a result data is passed by value rather than by reference. The language SISAL developed for the Manchester Dataflow Computer [GUR85] is an example of a language which employs single assignment. Tokens consist of data as well as a tag which identifies it and these actually migrate about within the system depending upon locality of execution. Execution occurs when all input tokens are available and their tags are

consistent. Upon execution a new token is created; thus data items are never modified.

Another example of a language employing the concept of single assignment is VAL [MCG82]. It uses a functional notation which prevents procedures from altering the program environment other than through their result. All data is passed by value.

1.3 The Goal of This Paper

The ACM model [UNG78] forms a basis for a language which employs the data flow principle. It is intrinsically concurrent, that is to say the programmer need have little if any conscious concern about the ordering of statements. Like CLU, objects are abstracted into a form which defines their usage. Both material and the "actions" which act upon the material are considered objects.

While basically driven by dataflow, control can be achieved through the use of stimulation and termination conditions, which together act much like guards. The success of a stimulation condition will enable a request (an "action" object to be executed) to fire but failure of the stimulation condition will not prevent it from being fired in the future. The success of a termination condition will absolutely terminate any associated request whether or not enabled.

The ACM model supports the notion of "partialing" which allows a request to become enabled before all of its material list is available. This feature corresponds closely to the "future" construct in MULTILISP. However there is no direct corresponding support for the "delay" construct although, through the use of stimulation conditions, this can be accomplished.

In the discussion of parallel processing and in dataflow languages in particular, the subject of granularity is often raised. Coarse-grained languages have substantial blocks of sequentially executed code running in parallel whereas fine-grained languages attempt to maximize

parallelism down to the smallest level. There are positives and negatives to each style: Coarse-grained languages are easier to implement and fit well with networks of processors, each with its own private memory store. Fine-grained languages require hardware designed to support them but may eventually provide the greater increase in processing speed. The level of optimal granularity is open to debate [GAJ85] and by and large depends upon the intended use and the architecture of the system. As the grain size decreases, more and more overhead is required for the scheduling and synchronization of the parallel fragments. For high performance maximal parallelism with the lowest possible overhead is desired. In general, it can be stated that languages based upon the data flow contain the potential for fine granularity. Since ACM is a dataflow model it contains the potential for fine granularity; however, its realization would require significant effort with hardware support that is unavailable at this time.

It is the intent of this work to extend the model to increase its flexibility and follow that with a proposal for implementation of a subset at an extremely coarse granularity level. It is more important, at this time, to concentrate on the language itself rather than upon its performance.

1.4 Summary of Contents

In this chapter an informal presentation of the problem being addressed was presented. This was followed by an overview of key ideas relating to the research area of parallel processing with particular emphasis on ideas being addressed within the paper. These ideas cover data abstraction, object orientation, control flow, and data flow. Following this was a brief description of the ACM model and how it relates to both the problem and the key ideas presented from current research. Chapter 2 will present a much more detailed look at the ACM model. In Chapter 3 several extensions will be presented for inclusion in the model to increase its flexibility. Chapter 4 will discuss the syntax and informal semantics of an implementation

with a description of limitations placed on the implementation. This may or may not contain extensions to the model presented in Chapter 3. Chapter 5 will discuss the actual implementation of the material presented in Chapter 4. Finally, Chapter 6 will summarize the accomplishments achieved, what was and was not included, and suggested directions for future research.

Chapter 2

2. The ACM model

2.1 Overview of the Model

The ACM model forms the foundation for an intrinsically concurrent prototype language where statements are executed in any order when the minimal requirements for their execution can be satisfied. These conditions primarily consist of the availability of the data input to them; hence the model conforms to the data flow principle. The programmer need have little knowledge of the means by which a problem is solved and instead can concentrate on the definition of the problem itself. As stated by [UNG78]:

"In order to program complex systems which involve concurrency, the computer scientist should learn to utilize lateral thinking. An initial step is to seek a lateral language which resembles familiar ones."

If the programmer needs to be concerned with the sequencing of activities, the flow of control can be expressly addressed through the use of boolean conditions placed upon statements in order to augment the basic data drive. As a result conditional execution is fully supported.

2.2 Objects

The model abstracts all objects into a unified structure which encompasses both material objects and the actions which can occur on them. An *object* consists of a five tuple whose components consist of *designator*, *attribute*, *representation*, *corporality* and *value*. Any object which lacks any of the components is incomplete. In order for any two objects to be equal, each of their respective components must also be equal.

The first component, *designator*, is a set of names which effectively identifies the object. The existence of the designator implies the existence of the object itself. The designator in turn consists of a three tuple whose components are the context, user name, and sequence. The context may be a set of names which further refine nested contexts of access. The user name is also a set of names by which the user references the data within the object either as a whole or in nested levels of subsets. The instance defines spatial position, chronological identity or monotonically increasing sequence number to identify specific subsets within the object. No more of the designator than is necessary need be specified to retrieve that portion of the object that is desired, as an example: a reference to an object with the user name "X" and for which sequence number is applicable might be "X" or "X..+0"; either is sufficient. Another example might be for the retrieval of the set of persons attending a seminar; the designator might be "seminar.attendees" whereas a further refined reference might have the user name "seminar.attendees.speakers".

The *attribute* component of the object defines the value type, internal structure, and relationship to other objects. The value type may be atomic or an aggregate consisting of further sub-objects. The internal structure defines the value type, particularly in the case of aggregates by defining the structure of the aggregate. The relationship of the object defines it in terms of its association with other objects in the system. The relationship, for example, of a data base record might be its commonality to other data base records of like type.

The 3rd component of an object, *representation*, defines the physical location of the object and its coding scheme.

The 4th component, *corporality*, defines the state of the object in terms of its longevity, the location of the "representation" component, replication count, and authorization of access.

Longevity takes on four values which are fixed, static, dynamic, and fluid with the ability to change increasing respectively. Fixed objects represent objects that exist prior to the model and are represented by decimal numbers (an example). Static objects may take on meaning but once and thus most fully mirror the concept of single assignment. Once "complete" a static object's components are permanently bound. Dynamic objects are allowed to have values which vary over time but for which separate "instances" are created. Finally, fluid objects may have their value change at any time between references. As objects pass from the context of their creation they may have their longevity change to a more restrictive type, but never to a less restrictive type.

Replication defines the copies of the object currently in existence as well as their availability for use. Corporality also defines the authorization of the object for use by the requester.

The last component of the object is *value* which may consist of a boolean, integer, real number or character as atomic types, or a set, collection, ordered collection or action as non-atomic types. The value of an object can not exist until all other components of the object exist.

2.3 Actions and Requests

An *action* can be defined as an object whose value defines some manipulation which can be performed on other objects. A *request* is the imperative statement of an action. The basic structure of both actions and requests is as follows:

$$(s, m, a, r, t)$$

Where the components of this five tuple consist of a stimulation condition, a material list, an action, a result list and a termination condition. The stimulation and termination conditions in an action are referred to as "internal" and those of a request, which augment the action, are

referred to as external. The material list represents those objects required (by the data flow principle) to be available before a request for an action can be enabled. The result list denotes those objects created as a consequence of the request having been performed. No result object is available until the request has completed and none are created unless the request ends successfully.

A request exists in various states, idle, enabled or disabled, depending upon various factors. These include basic data drive as well as boolean conditions placed upon the request in order to start it or terminate it. Generally states will transition from idle to enabled to disabled, although exceptions exist which allow other transitions to occur.

Let me introduce the following syntax in order to present some examples:

```
action([si] m1, m2, m3; r1, r2 [ti] );
```

Figure 2-1. Sample Action Declaration

Here an action specification is declared called "action" with has an internal stimulation condition call "si". It takes as input 3 material objects, "m1", "m2", and "m3", and produces 2 result objects, "r1", and "r2". If internal termination condition "ti" becomes true then the action will terminate. A request for this action might then be defined as follows:

```
[se] action( x1, x2, x3; y1, y2 ) [te];
```

Figure 2-2. Sample Request Declaration

The request for "action" will begin if material objects "x1", "x2", and "x3" are available and "x1" greater than 25 and condition "se" is true but condition "te" is false. Upon successful completion objects "y1" and "y2" have been created.

An action can be detailed into a sequence of sub-actions which, if permitted by the data flow principle, may run in parallel. This sequence is called the "request set" of the action, a simple example of which follows:

```
averages([occurrences > 0] counts, occurrences; a1, a2, a3)
{
    mean(counts, occurrences; a1) ;
    median(counts, occurrences; a2) ;
    mode(counts, occurrences; a3) ;
}
```

Figure 2-3. Detailing

A request for "averages" will be enabled when counts and occurrences are available and occurrences is greater than 0. The results are produced in parallel by the request set of the action. Within the context of a request for a detailed action, objects can be created and used as interim values for flow between members of the request set:

```
radius( x, y; z)
{
    square( x; x2 );
    square( y; y2 );
    add( x2, y2; x2y2 );
    squareroot( x2y2; z);
}
```

Figure 2-4. Locality of Context Within an Action Detail

A request for action "radius" will permit the squares of the material list to proceed in parallel followed sequentially by the sum of their results and the square root of the sum. The objects "x2", "y2", and "x2y2" are never seen outside the context of the request.

2.4 Control Constructs

Within the model flow of control can be applied using various constructs which augment the data drive. These constructs include *repetition*, *iteration*, *recursion*, *partialing*, *alternation*, and

caseation. *Alternation* and *caseation* are already well defined within the model through the use of stimulation conditions applied to the request set of an action. *Alternation* occurs when a pair of actions have stimulation conditions which are boolean opposites. Through the nesting of actions any combination of a conventional "if...then...else" construct can be created.

```
hightax( income; tax )
{
    [income <= 10000] multiply( income, .25; tax );
    [income > 10000] multiply( income, .50; tax );
}
computetax( income; tax )
{
    [income <= 1000] multiply( income, .10; tax );
    [income > 1000] hightax( income; tax );
}
```

Figure 2-5. Alternation

Caseation is simply the systematic application of mutually exclusive stimulations on a set of action details:

```
printerror( code; message )
{
    [ code = 1 ] assign( "error type 1"; message );
    [ code = 2 ] assign( "error type 2"; message );
    [ code = 3 ] assign( "error type 3"; message );
    [ code > 3 ] assign( "error unknown"; message );
}
```

Figure 2-6. Caseation

Repetition and *iteration* are constructs used to apply repeated application of a request over a range of values. *Repetition* occurs intrinsically with the model whenever the material and result lists specify objects which contain a range of values of like size, for example arrays. A separate and parallel invocation of the request will occur for each spatially matched pair. Given two arrays "a" and "b", each containing 3 elements the following request:

```
copy( a; b ) ;  
  
is equivalent to:  
  
copy( a(1); b(1) ) ;  
copy( a(2); b(2) ) ;  
copy( a(3); b(3) ) ;
```

Figure 2-7. Repetition

Iteration is a sequential reapplication of a request used when the results of one iteration must be used as material in the next iteration. For an iteration to occur an internal termination condition on the changing material/result pair must also be present. For example:

```
assign( 1, x ) ;  
assign( 1, fact ) ;  
factorial( x..+0, fact..+0; x..+1, fact..+1 [x < 16] ) ;
```

Figure 2-8. Iteration

The request "factorial" would be sequentially applied to objects "x" and "fact" so long as "x < 16". The iteration construct is an example of exceptional state transition where the enabled state returns to the enabled state.

Recursion occurs whenever an action's request set contains a request for that action. Each invocation creates a new and more limited context than the previous.

For actions which are detailed, *partialing* may be specified if the material list contains multiple items. This allows a request for the action to begin execution of the request set on those sub-actions which do not require the missing detail. For example suppose partialing was requested on the entire material list of the following action:

```
radius3( x, y, z; w )
{
    square( x; x2 );
    square( y; y2 );
    square( z; z2 );
    add( x2, y2, z2; x2y2z2 );
    squareroot( x2y2z2; w);
}
```

Figure 2-9. Partialing

If a request for this action occurs and only "x" and "y" are available, the request can proceed to calculate "x2" and "y2" because neither of these sub-actions require "z". The remaining sub-actions and thus the entire request will then "hang" waiting for the availability of "z", at which time the request can complete.

2.5 Summary

The ACM model forms the basis for a complete programming language which encourages a systematic block structuring approach through the detailing of "actions". Intrinsic parallelism is accomplished through the use of the data flow principle which is augmented by various control constructs which allow the programmer to specify conditional computation as appropriate. Data (and actions) has been abstracted into objects which shield the user from irrelevant details thus allowing for a concentration of effort on the task at hand.

Chapter 3

3. Analysis and Redefinition

3.1 Introduction

In this chapter we will look more closely at some of the characteristics of the ACM model and propose some extensions to the model in order to increase its flexibility.

3.2 An Extension to the Iteration Construct

An analysis of the original ACM definition of iteration has led to an extension of that definition in order to provide the user with greater flexibility. Consider the original definition of iteration:

Definition: An *iteration construct* is expressed as a single request in which $t_i \neq \phi$ in the action which is of the form

$$(s_i, m_i, a, r, t_i)$$

An iteration construct can be used to express the repeated requests

$$(s, m_i, a, r_i, t), \quad i=1, 2, \dots, n$$

where:

$$m_i = (m_{i1}, m_{i2}, \dots, m_{ij}), \quad j=1, 2, \dots, J$$

and

$$r_i = (r_{i1}, r_{i2}, \dots, r_{ik}), \quad k=1, 2, \dots, K$$

when the following conditions are met:

1. $m_{ij} = m_{xj}$ for all $i, x = 1, 2, \dots, n$ and for all $j = 1, 2, \dots, J$ except one $j = b$
2. $r_{ik} = r_{xk}$ for all $i, x = 1 \dots n$ and for all $k = 1, 2, \dots, K$ except one $k = c$
3. there exists a relation $m_{ib} = r_{(i-1)c}$ for all $n \geq i > 1$
4. the $m_{ib}, i = 0, 1, 2, \dots, n$ exist

5. there exists a termination condition $t = f(r_k)$.

Let us examine this definition for potential programming limitations by the presentation of some examples. Assume we wish to calculate the Fibonacci number sequence which has the following property:

$$x_0=1, x_1=1, \dots, x_i=(x_{i-1}+x_{i-2})$$

If we wish to calculate the first few numbers in this sequence we could state this in SMART (the informal syntax introduced in Chapter 2) as follows:

```
assign(1 ; x..0);
assign(1 ; x..1);
fibonacci(x..+0, x..-1 ; x..+1 [x..+0 > 3]);
```

Figure 3-1. Iteration Problem 1

Now let us analyze what occurs by looking at the material and result lists as the iteration proceeds:

```
m11=1 m12=1 r1=2 (x..2=2)
m21=2 m22=1 r2=3 (x..3=3)
m31=3 m32=2 r3=5 (x..4=5)
- termination -
```

Figure 3-2. The Material and Results of Iteration Problem 1

Note that by condition 1 we can observe that $m_{11} \neq m_{21} \neq m_{31}$ and therefore $b = 1$ but we also observe that $m_{22} \neq m_{32}$ and therefore $b = 2$. The conclusion reached is that any iteration requiring more than one of the items in the material list to change cannot occur under the original definition. Let us extend conditions 1 and 4 to allow this phenomenon:

1. $m_{ij} = m_{xj}$ for all $i, x = 1, 2, \dots, n$ and for all $j = 1, 2, \dots, J$ except a subset of j called X

where X is of cardinality α and $J \geq \alpha \geq 1$.
Let X_i denote set X between the i and $i-1$ iterations.

4. all elements in X_i exist for all $i = 1, 2, \dots, n$

Let us explore another example; assume we wish to calculate a number sequence consisting of the squares of the natural numbers: $f(x)=x^2$. We could state this in SMART as follows:

```
assign(0 ; x..0);  
powerseq(x..+0 ; x..+1, power..+1 [x..+0 = 3]);
```

Figure 3-3. Iteration Problem 2

Now again let us analyze what occurs by looking at the material and result lists as the iteration proceeds:

```
m11=0  r11=1  r12=1  
m21=1  r21=2  r22=4  
m31=2  r31=3  r32=9  
- termination -
```

Figure 3-4. The Material and Results of Iteration Problem 2

Note that by condition 2 we can observe that $r_{11} \neq r_{21} \neq r_{31}$ and therefore $c = 1$ but we can also observe that $r_{12} \neq r_{22} \neq r_{32}$ and therefore $c = 2$. The conclusion reached is similar to that of the last example, namely that any iteration requiring more than one of the items in the result list to change cannot occur under the original definition. Let us extend condition 2 to allow multiple result list items to change:

2. $r_{ik} = r_{jk}$ for all $i, x = 1, 2, \dots, n$ and for all $k = 1, 2, \dots, K$ except a subset of k called Y where Y is of cardinality β and $K \geq \beta \geq 1$. Let Y_i denote set Y between the i and $i-1$ iterations.

Recall that conditions 3 and 5 are predicated upon the assumption that there exist unique values for b and c which we have replaced in our extended conditions with sets X and Y respectively.

Therefore we must restate conditions 3 and 5:

3. for any iteration $i, i > 1, \alpha_i \leq \beta_{i-1}$ and $X_i \subset Y_{i-1}$
5. there exists a termination condition $t = \text{boolean } f(Y_i)$.

At this point we have a reasonably flexible definition of iteration which allows multiple items in both the material and results to change. Lest we become smug let us look at a somewhat bizarre example. Assume we wish to calculate a number sequence constructed as follows:

$$x_0=1, x_1=2, x_2=3, x_3=4, x_4=5, \dots x_i=(x_{i-5}+x_{i-3}+x_{i-1})$$

In we could state this in SMART as follows:

```
assign(1 ; x..0);
assign(2 ; x..1);
assign(3 ; x..2);
assign(4 ; x..3);
assign(5 ; x..5);
weirdseq(x..-4 , x..-2, x..+0; x..+1 [x..+0 > 120]);
```

Figure 3-5. Iteration Problem 3

Now yet again let us analyze what occurs by looking at the material and result lists as the iteration proceeds:

$m_{11}=1$ $m_{12}=3$ $m_{13}=5$ $r_{11}=9$
 $m_{21}=2$ $m_{22}=4$ $m_{23}=9$ $r_{21}=15$
 $m_{31}=3$ $m_{32}=5$ $m_{33}=15$ $r_{31}=23$
 $m_{41}=4$ $m_{42}=9$ $m_{43}=23$ $r_{41}=36$
 $m_{51}=5$ $m_{52}=15$ $m_{53}=36$ $r_{51}=56$
 $m_{61}=9$ $m_{62}=23$ $m_{63}=56$ $r_{61}=88$
 $m_{71}=15$ $m_{72}=36$ $m_{73}=88$ $r_{71}=139$
- termination -

Figure 3-6. The Material and Results for Iteration Problem 3

This does not even work with the revised iteration definition. Let us try to analyze why and generalize a solution to improve the flexibility of the definition.

First note that set X has a cardinal number α which must always exceed the cardinal number β of set Y ; that is to say there are always more items in the material list which change in each iteration than even exist in the result list. Clearly the changes in the material list must be coming from somewhere. They derive from multiple iterations. Let us introduce a new variable z which represents the maximum negative instantiation in the material list (for our example $z = 4$). Note now that any element in the set X must derive from no more than $z+1$ previous iterations. In the case where there are not that many previous iterations then any which do not derive from previous iterations must have preceded the entire iteration construct. We are ready to restate the iteration definition in its final form:

Definition: An *iteration construct* is expressed as a single request in which $t_i \neq \phi$ in the action which is of the form

$$(s_i, m, a, r, t_i)$$

An iteration construct can be used to express the repeated requests

$$(s, m_i, a, r_i, t), \quad i=1, 2, \dots, n$$

where:

$$m_i = (m_{i1}, m_{i2}, \dots, m_{ij}), \quad j=1, 2, \dots, J$$

and

$$r_i = (r_{i1}, r_{i2}, \dots, r_{ik}), \quad k=1, 2, \dots, K$$

when the following conditions are met:

1. $m_{ij} = m_{xj}$ for all $i, x = 1, 2, \dots, n$ and for all $j = 1, 2, \dots, J$ except for some subset of j called X of cardinality α where $J \geq \alpha \geq 1$
2. The set $X' = (U_{m_j} - X)$ is permanently bound at the invocation of the iteration. The longevity component of corporality changes, within the context of the iteration, to static.
3. $r_{ik} = r_{xk}$ for all $i, x = 1, 2, \dots, n$ and for all $k = 1, 2, \dots, K$ except for some subset of k called Y of cardinality β where $K \geq \beta \geq 1$.
4. Let X_i and Y_i represent the set of material and result items respectively which changed between the i th and $i-1$ st iterations. Let z represent the maximum negative instantiation in the material list. there exists a relation on $i, i > 1$ and z such that

- if $i \leq z$ then $\alpha_i \leq (\sum_{n=1}^{i-1} \beta_n) + z$

- if $i > z$ then $\alpha_i \leq \sum_{n=i-(1+z)}^{i-1} \beta_n$ and

$$X_i \subset \bigcup_{n=i-(1+z)}^{i-1} Y_n$$

5. all elements in X_i exist for all $i = 1, 2, \dots, n$
6. there exists a termination condition $t = \text{boolean } f(Y_i)$.

Where does this leave us? An informal summary of iteration is as follows: Any iteration construct must contain both material and result items which change between each iteration although no direct correlation can be drawn between the changing material of one iteration and any result preceding it. Material items which change must appear in the result list; those which do not change are bound at the start of the construct execution. The result list may contain

results which change but do not appear in the material list. Finally, an internal termination condition must be formed on some subset of the changing result list.

While this seems to be a sufficiently flexible definition, it is only fair to point out some remaining shortcomings which are not supported. No material items can change unless they appear in the result list. It would be nice to use the iteration construct in a manner which allows it to consume fresh material from the outside at each iteration. This is not possible under the current definition.

As a final note on iteration, I would like to propose an interesting extension which I will not develop beyond its description. An iteration depends upon the results of previous iterations; it forms an essentially sequential computation across its execution. If the concept of partialing can be applied to the changing material list set X , then substantial parallelism could be achieved. The iterating objects would need to be prebound to sequence numbers before the existence of their values in much the same manner that MULTILISP defines futures. The problem needed to be overcome in this case would be the reconstruction of what results exist if and when any arbitrary termination condition becomes true on an iteration.

3.3 The Deferred Execution Construct

While data drive is a fundamental concept in the ACM model, one construct missing from it which is closely related to data drive is a deferred execution construct. Such a construct would allow for the indefinite deferral of a request until such time as some item in its result list is required in another request which is otherwise ready to be enabled. In this case the data drive switches to demand mode. Deferring a request may be beneficial if there is some likelihood that the results of such a request may never be needed.

The model already contains the capability to perform this in a somewhat cumbersome and error prone manner as demonstrated in the following example:

```
[y2 exists]    process1( x1; y1 ) ;  
               .  
               .  
               .  
               process2( x2; y2 ) ;  
               process3( y1, y2; z3 );
```

Figure 3-7. Deferred Execution as Currently in the Model

The request "process1" is deferred until "process2", an otherwise artificial request, has completed and the existence of object "y2" is complete. Since "y1", the result of "process1" is used directly in "process3", the scheduling of "process1" will not occur until such time that "process3" would otherwise be ready to be enabled. Note however that this is not exactly deferred execution in that "process1" is not being stimulated on the need for an output but rather the availability of some other object. Any significant number of such artificial requests such as represented by "process2" would likely cause significant confusion (not to mention the extra overhead).

Within the model the corporality attribute itself contains a component called "availability" which is defined as a boolean which if true indicates the object as available and if false additionally contains information as to the reasons why the object is unavailable. If this component could contain deferral information in this field then a deferred execution construct would be possible. Note that the object being deferred is a request but the objects marked as deferred are the result objects that the request would produce. The format might look as follows:

```
[defer] process1( x1; y1 ) ;  
      .  
      .  
      .  
      process3( y1, y2; z3 ) ;
```

Figure 3-8. Potential Deferred Execution Construct

Here we make the keyword *defer* an external stimulation condition on the request to be deferred. The objects in its result list come into immediate existence but are marked unavailable. The availability component of the corporality would contain information denoting it as deferred and some logical connection to the request which can make it available. The demand for any result object in the deferred request would be sufficient to trigger the request (assuming it is not otherwise blocked by normal material needs or false stimulation conditions). The following is a more formal definition of the deferred execution construct:

Definition: A *deferred object* O_d is an object which exists and is not available and for which its non-availability is due to some request object R_d . The deferred status of an object is *removed* if and only if there exists some request R_n which would be ready to transition from the idle state to the enabled state except for some subset of its material list M_n which contains O_d and all elements in M_n which are not available are deferred. The *removal of deferred status from an object* sets the defer stimulus on R_d to true.

Definition: A *deferred request* is an object

$$(s_x, m, a, r_i, t)$$

when the following conditions are met:

1. The stimulation condition s_x consists of the set S of x boolean conditions for all $s_x \in S$, such that $s_1 \wedge s_2 \wedge \dots \wedge s_x$, forms the stimulation condition and $defer \in S$. The condition "defer" is initially false.
2. The set of result objects r_i are deferred objects.

3. The "defer" element of the stimulation set S is transformed to true if the deferred status of any object in r_i is removed.

3.4 Operations on Unordered Collections

The only operations allowed on unordered collections are ADD, DELETE, and REPLACE. It would be useful to add to this set the operations LENGTH and SUBLLENGTH. LENGTH reflects the number of elements in the unordered collection while SUBLLENGTH returns the number of elements of a particular type. Following is the original definition:

Definition: An *Unordered Set*. Let $[c_1, c_2, \dots, c_n]$ denote an unordered collection of objects; in particular, let $[\]$ denote the empty unordered collection ϕ .

Let $c^-[x]$ denote the unordered collection which results from concatenating the object x onto the end of unordered collection c

and

let D be the domain of the objects in an unordered collection.

Then:

- a) $[\]$ is an unordered collection, the null unordered collection ϕ .
- b) if c is an unordered collection, and $d \in D$, then $c^-[d]$ is an unordered collection.
- c) the only collections are those specified by a) and b).
- d) $ADD(c, [d]) =_{def} c^-[d]$
- e) $DELETE(c, [d]) =_{def} c$ if $d \notin c$
 or
 $DELETE(c, [d]) =_{def} c'$ if $d \in c$ where if the notation $[c_1, \dots, c_i, \dots, c_n]$ is an abbreviation for $[[c_1] \dots [c_n]] = c$ then if $d = [c_j]_{j=i, i_1, \dots, i_m}$ then $c' = [c_1, \dots, c_{[i-1]}, c_{[i+1]}, \dots, c_n]$

- f) $REPLACE(c, [d], [e]) =_{def} c$ if $d \notin c$
 or
 $REPLACE(c, [d], [e]) =_{def} c'$ if $d \in c$ where if
 $d = [c_j], j=1, i_1, \dots, i_m$ then
 $c' = [c_1, \dots, c_{[i-1]}, e, c_{[i+1]}, \dots, c_n]$

Now in order to add the LENGTH and SUBLENGTH operators we must add the following to the above definition:

- g) $LENGTH([]) =_{def} 0,$
 $LENGTH(c, [d]) =_{def} 1 + LENGTH(c)$
 h) $SUBLENGTH(c, [d]) =_{def} 0$ if $d \notin c$
 or
 $SUBLENGTH(c, [d]) =_{def} 1$
 $+ SUBLENGTH(c', [d])$
 if $d \in c$ and $c' = c - [d]$

3.5 Mapping Coercion in Repetition Constructs

What if we wish to invert the order of elements in an array and store it in another array? The repetition construct is designed to allow parallel actions across the ordered values of an object.

The mapping of material objects to the result objects is defined as follows:

Definition: The mapping ζ defined on the linear order of the structures involved. Given two structures M and R with linear order $M=(m_1, m_2, \dots, m_j)$ and $R=(r_1, r_2, \dots, r_j)$ then the mapping of M to R is $m_i \zeta r_i$.

The linear order is established from the spatial coordinates of a structure or from the order defined within an abstraction. The mapping ζ is otherwise undefined.

Three cases exist to determine linear order:

- a) The linear order for structures with spatial coordinates is defined in the model such that elements are ordered by letting the first coordinate vary most rapidly, the second varies next most rapidly, etc.

The remainder of the definition deals with ordered collections and is not relevant to this extension.

Now consider the following repetition construct where the action being requested inverts the elements of an array, an object which uses spatial coordinates:

invert(a; b);

By the above definition of mapping the only mapping allowed is

$a[1] \rightarrow b[1], a[2] \rightarrow b[2], \dots, \text{etc.}$

In this example we wish to really map

$a[1] \rightarrow b[n], a[2] \rightarrow b[n-1], \dots, \text{etc.}$

How can we adjust the model to allow this, or any other arbitrary linear mapping, to occur?

The object component "attribute" defines the logical organization of the object. Using this component we can make the following definition:

Definition: A *mapping object* is a complete object (d, a, r, c, v) such that:

1. The designator d exists.
2. The attribute a exists and defines a linear function on the natural numbers N such that $\xi_i = f(N_i)$
3. The representation r exists but is not relevant (vacuous).
4. The corporality c exists as static and available.
5. The value v exists and is the null set.

The restriction to linear functions is required so that no mapping to the same natural number can occur more than once. Using the above definition we can restate case a in the definition of *mapping* as follows:

- a1) The linear order for structures with spatial coordinates is defined in the model such that elements are ordered by letting the first

coordinate vary most rapidly, the second varies next most rapidly, etc.

- a2) The linear order for structure with spatial coordinates may be coerced to use the linear functional value ξ of a mapping object for any coordinate.

The semantic definition of a mapping object is beyond the scope of this paper; let us assume one exists for "inverse". Returning to our previous example the use of a mapping object might appear as follows:

copy(a(invert); b);

The effective mapping would result in $a_i \xi b_i$. The mapping of any array to another can occur in any linearly definable fashion. Each spatial coordinate of a multiple dimension array can be treated individually producing a wealth of possible transformations.

3.6 A Summary of Other Research

The following extensions are being researched at this time in parallel with this effort. They will be briefly mentioned here for completeness. In [YUK86] the following extensions were proposed:

- A restatement of request states and state transitions to eliminate some ambiguities and define the concept of unsuccessful completion of a request.
- The addition of boolean predicates which allow the success or failure of a request to be used in the stimulation or termination of other requests.
- An pre-defined action which forces a request to fail.
- The definition a default case for the caseation construct.
- The addition of the concurrent alternative construct which allows parallel execution of

alternative requests.

Additionally the following were proposed by [SHI87]:

- A indefinite looping construct which rebinds the context of its material and result lists at each iteration.
- The inclusion of a dynamic FIFO data object which is particularly useful in the indefinite looping construct in order to solve producer/consumer problems.

Chapter 4

4. SMART - an ACM Language (the Semantics)

4.1 Introduction

A thorough discourse on the semantic meaning of any language, even a simple one, is admittedly a difficult chore. Anything less risks the danger of raising more questions than it answers; this chapter faces that risk. It contains a rather informal presentation of the semantics of the implementation of SMART. The choices made were not well developed beyond the simple expediency of selecting a meaning which was consistent with what was to be accomplished within the limited implementation. As a result most of the semantics discussed here are probably arguable, particularly if applied to a more general implementation.

4.2 Limitations on the Implementation

While a more thorough discussion of the implementation can be found elsewhere [SHI87], it would be helpful to briefly state the limitations placed upon the implementation of SMART. In brief:

- Data object types have been limited to the atomic types *integer*, *real*, and *character*.
- Longevity of data objects can be *static*, *dynamic*, *fluid*.
- No aggregations or spatial coordinates (arrays) are allowed.
- No detailing of actions occur. All requests are for essentially atomic actions which internally behave outside of the model.
- No type checking occurs; the assumption used is that whatever atomic action is requested will be able to use the material data object types as presented and will produce exactly the data

object types specified in the results.

- Conditions are formed from any valid grouping of boolean operations.
- Conditions may contain the request success/failure extension described in [YUK86].
- The only major construct available is the extended version of iteration presented in chapter 3.

Note that alternation and caseation can be simulated through the judicious use of external stimulations.

4.3 Informal Semantics

This section will cover the static semantics of SMART but will occasionally lapse into dynamic semantics. The intent is to provide the user enough information in order to use the implementation. It is assumed that the reader is familiar with the syntax of SMART; a discussion of this subject can be found in [SHI87]. Additionally the BNF for SMART is in the attachments.

Data objects must be declared with the keyword *var* followed by a longevity keyword, *static*, *dynamic*, or *fluid*, followed by the type, *int*, *real*, or *char* and finally followed by an object designator or list of designators. For example:

```
VAR  STATIC      int    x, y;
      FLUID       real   f1, f2;
      DYNAMIC     int    a ;
      char       char   b;
```

Figure 4-1. Data Object Declarations

All of the above are valid declarations. If the longevity is not specified, as in the case of "b", then the default of DYNAMIC is assumed. Reference to data objects in subsequent requests is limited to the user name component of the designator. In the case of DYNAMIC objects the

designator may have a relative or absolute sequence number associated with it.

With the declaration of a static data object a value may be assigned to it from the universe of *fixed* objects, for instance:

```
VAR  STATIC      int    x = 1,  
     STATIC      char   a = 'a';  
     STATIC      int    y;  
     STATIC      char   b;
```

Figure 4-2. Data Object Value Initialization

The first 2 objects, "x" and "a", have initialized values. Because they are static they cannot be changed in value whereas "y" and "b" may change due to some later inclusion in a result list since they are incomplete (no value component).

Because no detailing is being done at this time, the statement of any request implies the action being requested must be atomic; therefore a request may appear as follows:

```
label: [se] action(m1, m2, ... mj ; r1, r2, ... rk [ti]) [te] ;
```

Figure 4-3. A General Request Declaration in SMART

This is the most expansive request object which can be used in the implementation. A label may be provided for use in the stimulation or termination conditions of other requests. Optional external stimulation and termination conditions can be formed on any boolean expression. Because no detailing is allowed internal termination conditions are allowable within the request in order to facilitate iteration (the one major construct being implemented). Finally the material and result lists, separated by a semi-colon, may contain multiple objects and reasonable (and as yet undetermined) sizes of these lists are acceptable. White space is optional anywhere and all statements are terminated by a semi-colon. Stripped to its simplest form a request appears as follows:

```
action( m ; r ) ;
```

Figure 4-4. A Simple Request Declaration in SMART

A simple request consists of the atomic action with a statement of at least 1 material object and 1 result object and with no conditions specified. Because the atomic actions exist outside of the model, even this simple form may in fact have no material and/or result lists. Simple requests, as well as any other, are entirely driven by data flow; no explicit ordering of requests adds any imperative ordering to the execution.

The specification of a dynamic data object within a material or result list may occur with or without sequence numbers. Within the material list a dynamic object defaults to the latest value, while within the result list it is the next value in the sequence. Negative relative sequence numbers are allowed in a material list whereas they are not allowed in the result list. Similarly a positive sequence number may be applied (limited to "..+1") to a result object but not to a material object. Absolute sequence numbers may also be used. An example of the correct usage of dynamic objects follows:

```
VAR DYNAMIC int x, y, z;  
multiply(x, y..-1 ; z..+1)
```

Figure 4-5. Correct Use of Dynamic Data Objects

Here the latest sequence of "x" and the second latest sequence of "y" are used to produce the next sequence of "z". Any sequence of "x" will suffice but at least 2 sequences of "y" must be available before the data drive will allow this request to be enabled. An example of the incorrect usage of dynamic sequenced objects is:

```
VAR DYNAMIC int x, y, z;  
    STATIC int a, b;  
  
zeppelin(a..+0, x..+1, y..0 ; b, z..+0) ;
```

Figure 4-6. Incorrect Use of Dynamic Data Objects

This example will go down in flames; a number of problems exist.

- Since "a" is a static object it cannot have a sequence number.
- The dynamic object "x..+0" used here would require that the next version after the latest be available before the request can be enabled. When that value became available it immediately becomes designated "x..+0", therefore "x..+1" would never be available.
- "z" produces the latest sequence that already exists, a violation of single assignment.

Iteration adds some restrictions to the material and result lists; They must contain at least 1 common dynamic data object and of those in common at least one must be a part of a required internal termination. A valid example is as follows:

```
VAR DYNAMIC int x, y, z;  
    FLUID int a;  
  
iter(a, x, y ; x, a, z [x < 20]) ;
```

Figure 4-7. Iteration

Here material "a", "y" and "x" are used to produce "x", "a" and "z". Since "x" is the only dynamic object in both the material and result lists it must also appear in the internal termination. Even though "a" appears in both the material and result lists it is fluid and thus not an iterating value.

The context of a request is bound when the request is enabled. In the above example should "y" be unavailable and some other request is producing a sequence of "x" values the binding of which "x" is used as input to this request will not occur until "y" becomes available. Furthermore for

each iteration, even though "a" and "y" are fluid and dynamic respectively, the values bound in each iteration remain the same as at the outset. Since "a", "x", and "z" appear in the result list they are only available to any other request at the termination of the entire iteration (of course pre-existing values are available). Note that since "a" is fluid only one value is produced but since "x" and "z" are dynamic the number of values produced are entirely dependent upon the number of iterations that occur.

The essential difference between an internal termination and an external termination, outside that of iteration, is the context in which it is evaluated. Consider the following:

```
VAR FLUID x;  
action(x, y; z [x < 5]) [x > 10];
```

Figure 4-8. Internal Versus External Termination

The above is not an iteration because no material list objects appear in the result list. It would appear that the action will terminate if "x<5" or "x>10". This is true at the outset because when the context of the action is established both conditions are evaluated. However, once enabled the contextual binding effectively eliminates any further need to examine the internal termination as it can no longer change. The external stimulation can change because outside of the context of the action "x" is still capable of change through other requests. At each change in value the external stimulation must be reevaluated. If "x" was dynamic rather than fluid this would still be true, however, were "x" static then then both the internal and external stimulation both would need only be evaluated once, when "x" is available.

The success of a request implies that all of its results have been produced and the failure implies that none of its results are produced. The success or failure of a request can be used in the simulation or termination conditions of another request; for example:


```
producer(a; x, y);  
ave1: average1(x, y, z) [S(ave2) | S(ave3)];  
ave2: average2(x, y, z) [S(ave1) | S(ave3)];  
ave3: average3(x, y, z) [S(ave1) | S(ave2)];
```

Figure 4-9. Success or Fail within Conditions

In this example the 3 requests, "ave1", "ave2" and "ave3" can all be started in parallel based upon the data drive, but only one can succeed because any success immediately results in failure of the others. Only 1 value "z" will be produced.

Finally, let us look at the evaluation of stimulation and termination conditions.

```
job1(x, y, z);  
[x<z] job2(x, y, z ;a);  
job3(x, y, z; b) [a<5];  
job4(x, y, z; c) [(z<3)(x<y)];
```

Figure 4-10. Condition Evaluation

Here "job1" is a simple request which is enabled as soon as its material list is available. Upon completion "job2", "job3", and "job4" are ready to be enabled by pure data drive but each has a condition which needs evaluation. "job2" will transition to enabled if "x<z" is true. If this condition is not true then "job2" remains idle until such time as "x<z" becomes true or until the program terminates. At least one of "x" and "z" must not be static for this condition to even have the potential of change. "job3" can be enabled immediately even if "a" has no value because a condition on an incomplete value is considered false and, therefore, the completion of "job3" may occur even if "a<5" since it may complete before "job2" produces an "a" to evaluate. If "job3" is either idle or enabled (and running) it will be terminated if "a<5" becomes true.

If a condition contains an incomplete object only the expression using the object is false, so in fact the condition as a whole might still evaluate as true. Consider the termination condition of

"job4"; even if "z<3" is false because "z" is incomplete, if "x<y" is true then the termination still occurs. "job4" does not have the problem that "job3" represents because its termination condition depends upon "x", "y" and "z" being available. Since the data drive for "job4" also depends on these objects there is no way for "job4" to run with an indeterminate condition. As a final point note that any time a termination condition becomes true the associated request transitions to the disabled state.

Chapter 5 .

5. SMART - an ACM Language (The Implementation)

5.1 Introduction

This chapter will discuss the implementation of the basic SMART node and how it is used.

5.2 Compilation

Programs for SMART must be compiled before they can be executed within the node. Further information on the compiler can be found in [SHI87].

5.3 The SMART Network

The SMART network consists of a variable number of nodes which communicate across an ethernet link between up to as many 5 systems (no reason exists why more could not be specified). The utilization of any node by any other node is entirely transparent to the user. Each node on the network is capable of servicing requests originating locally (new executable programs) or remotely (tasks for remotely controlled programs). The structure of the network is thus simple and can be represented by the following diagram:

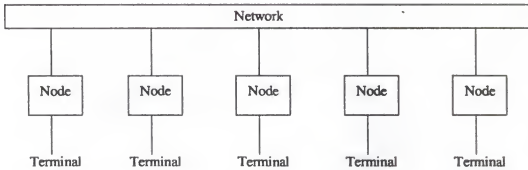


Figure 5-1. The SMART Network

Each node represents a set of interacting processes which can also communicate interactively with a local terminal as well as as other node which is up.

5.4 The SMART Node

A SMART node consists of 5 basic control modules and a library of executable atomic actions. The basic modules consist of the *node manager*, *terminal driver*, *scheduler*, *network driver*, and *environment handler*. They form a set of communicating sequential processes which interface using message queues. A diagrammatic representation of the basic module interconnections is as follows:

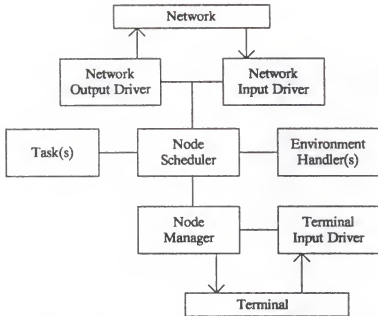


Figure 5-2. The SMART Node

The *node manager* controls the node by starting the other basic modules, routing terminal communications to the appropriate place, logging information to a file (when requested) and terminating the node (when necessary). All requests to the node manager originate from its private input message queue and the activity performed varies:

- Messages arriving from the terminal are checked for direct commands to the node manager. These include logging comments, killing the node, reprinting outstanding questions, and killing active programs.
- Messages arriving from the terminal which are not direct commands are parsed to determine if they represent responses to questions for which other modules are awaiting replies. Such messages are routed to the appropriate module through a common message queue.
- Messages from the terminal which do not correspond to the above categories are assumed to

be requests for new programs and are forwarded to the scheduler via its private message queue.

- Messages arriving from other than the terminal will either be logged, displayed on the terminal, or both. Messages to the terminal which represent questions will be stored for later reference when the response from the terminal arrives.

The *terminal driver* simply spools terminal input into messages (delineated by a newline) and passes the packetized request to the node manager. White space at the beginning of a new line is thrown away and '@' is treated as a line kill character. One other special character, '|', allows the user to escape to the terminal shell in order to accomplish non-SMART related commands. As a final point, the terminal driver is the only process in the SMART node that does not ignore a <brk>, which it uses to construct a "kill node" command for the node manager.

The *scheduler* controls the execution of tasks (requests in process), the execution of environment handlers (programs in process) as well as the locality of tasks within the network. It also is driven by input on a private message queue and depending upon the message type performs the following activities:

- A new program request arriving from the terminal will result in the creation of a new environment handler to control it.
- Requests for tasks arriving from local environment handlers will result randomly in the local execution of a task, or in a remote task request to another node.
- Requests for tasks arriving from other nodes will result in local execution of such tasks.
- The successful or unsuccessful termination of tasks will result in the routing of the status to the appropriate environment handler, either locally or remotely.

- If another node goes down any tasks located on that node are automatically rescheduled somewhere else.
- A request to kill an environment handler (from the terminal) will result in the termination of that handler as well as any local or remote tasks in progress for it.
- A display of the current local environments and tasks in progress will occur if requested. A running display of activity is also available for debugging programs.

The *network driver* is responsible for the interface to other nodes via the ethernet. Incoming packets are routed to the scheduler and outgoing messages are sent to the proper node. More details on the network driver are to be found in [YUK86].

The *environment handler* forms the true executable portion of the SMART language. It parses the dataflow graph and forwards enabled requests for execution through the scheduler (as tasks). The arrival of a task completion message will contain the status of the request as well as the results. Further discussion of the environment handler is beyond the scope of this document (see [YUK86]).

5.5 The SMART Library

Additionally the SMART node contains a library of atomic actions which can be used as requests. These execute as tasks and must conform to the standard message handling interface using the common message queue for any and all terminal communication. Upon completion they pass a task done response message back to the scheduler. A list of the atomic actions which are currently available may be found in the attachments.

5.6 User Instructions

A SMART node is started by entering the "smart<cr>" command. The user will be immediately queried as to the node number, which must be unique. Each node must be started individually and will automatically connect to any other active nodes in the network. No requirement exists for more than one node to be up in order to use that node. If a node goes down any work on that node being performed for another node will automatically restart elsewhere.

If the node needs information from the terminal a message will appear on the terminal prefixed by a "nn: " where "nn" is a sequence number. The node itself does not wait for the reply but instead continues processing whatever work it has ready. This may include additional questions to the terminal. When the user wishes to reply the answer must be prepended with "nn: " where "nn" is the same sequence number of the original message. Should the outstanding messages be forgotten the user may enter "?<cr>". This will result in the re-display of all outstanding questions.

In order to start a program the user must enter "name<cr>" where name is the resultant output of the successful execution of the SMART compiler. Anything entered at the terminal which cannot be determined to be another command will be assumed to be a request for a new program. Whenever a new program is started the process number of the environment handler for that program will be displayed.

In the case where the terminal user wishes to stop a particular program the command "kill nnnn<cr>" where "nnnn" is the process number of the environment handler will result in the termination of that handler along with any currently active tasks, either local or remote.

Entering "list all<cr>", "list jobs<cr>", or "list tasks<cr>" will display the current status of

activity within the node. Entering "list active<cr>" will display each task request processed as they occur; "list silent<cr>" will log each task as they occur, and "list stop<cr>" will stop an active or silent display.

The state of an active program can be logged by entering "dump nnnn<cr>" where "nnnn" is the process number of the environment handler for the program. When a program terminates a message indicating "done" or "hung" will be displayed based upon whether or not any requests did not transition to the disabled state.

The node itself may be terminated at any time by entering "exit<cr>" or "<brk>". This will result in the orderly shutdown of the node allowing other nodes to continue to function correctly.

Finally the SMART session will result in a record of the activity on that node in a file called "logfile". Additional information on using the smart node is found in the attachments.

Chapter 6

6. Conclusions

6.1 Accomplishments

The problem addressed by this paper has been the task of freeing the programmer from the difficulties encountered when dealing with concurrent processing. The need for such processing is becoming more acute as the maximum threshold for performance on traditional computer architecture is rapidly being approached. Relevant ideas have been presented from current research in the area of parallel processing with particular emphasis on data drive models. Of particular interest are the concepts of:

- data abstraction
- object orientation
- the data drive principle
- single assignment
- the elimination of side effects.

Using the ACM model as a basis for a language employing the data drive principle, several extensions to the model were proposed:

- An expanded definition of iteration which allows significant flexibility in its use.
- A deferred execution construct which allows the model to run in a demand mode, thus deferring potentially forever the unneeded overhead of unnecessary activity.
- Extensions to the build-in operations on unordered collections which allow the size of the

collection and sub-collections to be determined.

- A definition of a mapping object which allows a repetition construct to associate in any way objects of different spatial design.

An implementation of a modest subset of the ACM model was done, hopefully as a building block to a more complete implementation in the future. In order to maintain the data drive principle of inherent concurrency this implementation was designed to perform on multiple processors connected in a network where enabled requests are performed at random anywhere within the complex.

6.2 Notable Limitations

Although the extended definition of iteration adds significant flexibility it still retains several shortcomings. No provision is made for the inclusion of new material to the construct after its invocation except for the interesting data objects already contained within the context of the iteration.

Additionally if the concept of partialing could be extended to the result list a significant performance enhancement might be achieved in parallel execution. As it currently exists an iteration has absolutely no internal parallelism, but one can easily construct, at least informally, many examples where parallelism within the iteration would have significant benefit. The difficulty in extending the definition involves defining exactly what exists when any given iterate, assuming many running concurrently, has its termination condition satisfied.

The introduction of a mapping object was a significant improvement to the repetition construct. Unfortunately it is difficult to envision exactly how such an object can be syntactically defined in order to use it.

The implementation of SMART is notably incomplete in that only the iteration construct was even attempted. Of particular interest would have been detailing of actions which would have lent a block structured flavor to the language. In addition the inclusion of both files and aggregates would have made the language more useful.

6.3 Areas for Future Research

Directions for future research in the use of the ACM model is a fertile field as the model provides a rich diversity for the definition of objects. While it is not hard to come up with ideas I would suggest several which should come as no surprise:

- The extension of partialing to the result list probably would have significant performance enhancement possibilities.
- The syntactic and semantic description of a mapping object to make this a useful tool.
- A definition of file handling within the model would take the model a long way to a legitimate implementation.

The implementation of SMART has significant limitations, the elimination of any one would improve the language and its usefulness. As previously stated the most significant limitation was the lack of detailing. Additionally the implementation of constructs, aggregate objects, files, and the use of contextual designators; all of these would enhance the language. No particular emphasis was made on the performance of the implementation; this would also be an area well worth study. Although not really a part of the model nor the language, the current smart node has the capability to allow multiple work station interaction. It would be a simple effort to extend the language to allow directed task execution to specific nodes.

As a final point, ACM, a data flow model, would function well on a hardware system designed for data flow. It would be nice to see it implemented for such hardware.

SMART User Notes

The SMART Node: 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.

The SMART Library

The SMART Library

The following list represents the atomic actions currently available:

Demonstration Actions:

askadd, asksub, askmult, askdiv

Each of these require 2 integer materials and produce 1 integer result. The terminal connected to the node upon which the task is run determines will be queried for a response.

Integer Math:

add, sub, mult, div

Each of these require 2 integer materials and produce 1 integer result.

Real Math:

addf, subf, multf, divf

Each of these require 2 real materials and produce 1 real result.

SMART BNF SYNTAX

The SMART BNF Syntax

```
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
            |   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
```

SMART Sample Program 1

Smart Sample Program 1

```
/*
 * Solve simultaneous equations:
 *
 *      (a1 * x) + (b1 * y) = c1
 *      (a2 * x) + (b2 * y) = c2
 */

VAR  static real    a1;    /* 1st line values */
      static real    b1;
      static real    c1;

      static real    a2;    /* 2nd line values      */
      static real    b2;
      static real    c2;

      static real    x;    /* intersection point answer */
      static real    y;

      static real    xinter1; /* alternate variables      */
      static real    xinter2;
      static real    yinter1;
      static real    yinter2;
      static real    same1;
      static real    same2;
      static real    same3;

      static real    det;
      static real    det1;
      static real    det2;

      static real    val1;
      static real    val2;
      static real    val3;
      static real    val4;
      static real    xval;
      static real    yval;

/*
 * the following represents the most common solution
 */
multf(a1, b2, det1);
multf(a2, b1, det2);
```

```
    subf(det1, det2; det);

    multf(b2, c1; val1);
    multf(b1, c2; val2);
    multf(a2, c1; val3);
    multf(a1, c2; val4);

    subf(val1, val2; xval);
    subf(val4, val3; yval);

div1:  divf(xval, det; x);
div2:  divf(yval, det; y);

ans1:  print("lines intersect at ", x, y) [F(div1) | F(div2)];

/*
 * the following checks y intercepts
 */
yint1: [F(div1) | F(div2)]divf(c1, b1; yinter1);
yint2: [F(div1) | F(div2)]divf(c2, b2; yinter2);

an2:   [xinter1=xinter2]print("same vertical line")[S(an4)];
an3:   [(xinter1!=xinter2) & (F(yint1) | F(yint2))]print("parallel vertical lines");

/*
 * the following checks x intercepts
 */
xint1: [F(div1) | F(div2)]divf(c1, a1; xinter1);
xint2: [F(div1) | F(div2)]divf(c2, a2; xinter2);

an4:   [yinter1=yinter2]print("same horizontal line")[S(an2)];
an5:   [(yinter1!=yinter2) & (F(xint1) | F(xint2))]print("parallel horizontal lines");

/*
 * check for same line
 */
divf(a1, a2; same1);
divf(b1, b2; same2);
divf(c1, c2; same3);
an6:   [F(ans1) & (same1=same2) & (same1=same3)]print("same line");
an7:   [F(ans1) & (same1=same2) & (same1!=same3)]print("parallel lines");

/*
 * get the input
 */
l1:    print("solve simultaneous equations");
l2:    [S(l1)] print("    (a1)x + (b1)y = c1");
```

```
13: [S(12)] print(" (a2)x + (b2)y = c2");
14: [S(13)] prompt("enter real a1"; a1);
15: [S(14)] prompt("enter real b1"; b1);
16: [S(15)] prompt("enter real c1"; c1);
17: [S(16)] prompt("enter real a2"; a2);
18: [S(17)] prompt("enter real b2"; b2);
[S(18)] prompt("enter real c2"; c2);
```

SMART Sample Program 2

SMART Sample Program 2

```
/*
 * test dynamics across several iterations
 * (fibonacci number sequence)
 */

VAR static int one=1;
    static int two=2;
    dynamic int x;
    dynamic int max;

/*
 * initial setup
 */
prompt("enter max number"; max);
a1: assign(one; x);
[S(a1)] assign(one; x);
    mult(two, max; max);

/*
 * iterate until complete
 */
f1: add(x..+0, x..-1; x..+1 [x..+0 > max..-1]);
f2: [S(f1)] add(x..+0, x..-1; x..+1 [x..+0 > max..+0]);

/*
 * print the last 12 numbers in the sequence
 */
p1: [S(f2)] print("last 12 answers: ", x..+0);
p2: [S(p1)] print(" ", x..-1);
p3: [S(p2)] print(" ", x..-2);
p4: [S(p3)] print(" ", x..-3);
p5: [S(p4)] print(" ", x..-4);
p6: [S(p5)] print(" ", x..-5);
p7: [S(p6)] print(" ", x..-6);
p8: [S(p7)] print(" ", x..-7);
p9: [S(p8)] print(" ", x..-8);
pa: [S(p9)] print(" ", x..-9);
pb: [S(pa)] print(" ", x..-10);
[S(pb)] print(" ", x..-11);
```

Bibliography

Bibliography

- [ACK79] Ackerman, W.: "Data Flow Languages", Proceedings of the National Computer Conference, Volume 48, 1979
- [ALM85] Almasi, G.: "Overview of Parallel Processing", Parallel Computing (North Holland), Vol 2, 1985
- [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
- [BOR82] Boral, H. and Dewitt, D.: "Applying Data Flow Techniques to Data Base Machines", IEEE Computer, August 1982
- [BRA84] Bracchi, G. and Pernici, B.: "The Design Requirements of Office Systems", Acm Transactions on Office Information Systems, Vol 2, No 2, April 1984
- [COM76] Comte, D., et al.: "Systeme LAU", ONERA CERT - Department of Informatics, TEAU 9/7, 1976
- [COM77] Comte, D., et al.: "Parallelism, Control and Synchronization Expression in a Single Assignment Language", ONERA CERT - Department of Informatics
- [DAV82] Davis, A. and Keller, R.: "Data Flow Program Graphs", IEEE Computer, Feb 1982
- [DEN69] Dennis, J.: "Programming Generality, Parallelism and Computer Architecture", Information Processing 68; North-Holland, 1969
- [DEN72] Dennis, J.: "First Version of a Data-Flow Procedure Language", Project MAC Computational Structures Group Memo 93-1, MIT, 1972
- [DEN80] Dennis, J.: "Data Flow Supercomputers", IEEE Computer, Nov 80
- [DER76] Deremer, F. and Kron, H.: "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Transactions on Software Engineering, Vol. 2, No. 2, June 1976
- [DIJ75] Dijkstra, E.: "Guarded Commands, Non-Determinacy and a Calculus for Definition of Programs", Communications of the ACM, Vol. 17, No. 8, 1975
- [GAJ82] Gajski, D., et al.: "A Second Opinion on Data Flow Machines and Languages", IEEE Computer, Feb 1982
- [GAJ85] Gajski, D. and Jih-Kwon PEIR: "Comparison of Five Multiprocessor Systems", Parallel Computing (North-Holland), Vol 2, 1985
- [GAN75] Gannon, J.: "Language Design for Programming Reliability", IEEE Transactions on Software Engineering, Vol. 1, No. 2, June 1975
- [GUR85] Gurd, J., et al.: "The Manchester Prototype Dataflow Computer", CACM, Jan 1985
- [HAL85] Halstead Jr., R.: "Multilisp: A Language for Concurrent Symbolic Computation", Acm Transactions on Programming Languages and Systems, Vol. 7, No. 4, Oct 1985

- [HAN75] Hansen, P.: "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol. 1. No. 2, June 1975
- [HAR85] Harris, J.: "Representing and Effecting Parallelism in Programs for Control-Flow Multiprocessors: Program Graphs and a Program Graph Interpreter", Masters Thesis - University of Illinois, Jan 1985
- [HIB78] Hibbard, P., et al.: "A Language Implementation Design for a Multiprocessor Computer System", Proceedings of the Fifth Annual Symposium on Computer Architecture, 1978
- [HOA78] Hoare, C.: "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, August 1978
- [HUN86] Hunt, J.: "Detection of Deadlocks in Multiprocess Systems", SIGPLAN Notices, Vol 21, No 1, January 1986
- [KOG85] Kogge, P.: "Function Based Computing and Parallelism: A Review", Parallel Computing (North-Holland), Vol 2, 1985
- [LIS77] Liskov, B., et al: "Abstraction Mechanisms in CLU", Communications of the ACM, Vol. 20, No. 8, August 1977
- [MCG82] McGraw, J.: "The VAL Language: Description and Analysis", ACM Transactions on Programming Languages and Systems, Vol 4, No 1, Jan 1982
- [MOT85] Motteler, H. and Smith, C.: "A Complexity Measure for Data Flow Models", International Journal of Computer and Information Sciences, Vol 14, No 2, 1985
- [OHB85] Ohbuchi, R.: "Overview of Parallel Processing Research in Japan", Parallel Computing (North-Holland), Vol 2, 1985
- [PER79] Perrott, R. "A Language for Array and Vector Processors", ACM Transactions on Programming Languages and Systems", Vol. 1, No. 2, October 1979
- [PET77] Peterson, J.: "Petri Nets", Computing Surveys, Vol 9, No 3, Sept 1977
- [SHI87] Shirley, E.: "A Programming Language and Compiler Design Based on an Augmentation of the ACM Model", Masters Thesis, Kansas State University, 1987
- [STO??] Stotts, P: "A Comparative Survey of Concurrent Programming Languages", University of Virginia
- [UNG78] Unger, E.: "A Concurrent Model" Doctoral Dissertation, University of Kansas, 1978
- [WAT82] Watson, I. and Gurd, J.: "A Practical Data Flow Computer", IEEE Computer, Feb 1982
- [WIL84] Williamson, R. and Horowitz, E.: "Concurrent Communication and Synchronization Mechanisms", Software Practice and Experience, Vol. 14, No. 2, Feb. 1984
- [WOO83] Woo, N. and Agrawala, A.: "The DC1 Flow Schema with the Data/Control-driven Evaluation", Proceedings of the International Conference on Parallel Processing, 1983
- [YUK86] Yuknavech, R.: "An Implementation of the ACM Dataflow Model" Masters Thesis, Kansas State University, 1986

Extensions to the ACM Dataflow Model

by

Robert Wayne Fish

B. S., University of Denver, 1969

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

1987

Abstract

This paper addresses the problem of freeing the programmer from the difficulties encountered when dealing with concurrent processing. The need for such processing is becoming more acute as the maximum threshold for performance on traditional computer architecture is rapidly being approached. Relevant ideas are presented from current research in the area of parallel processing with particular emphasis on data drive models. An overview of ACM, a data flow model, is presented and a number of extensions to the model are proposed. Finally, the description of an implementation of a limited subset of the model is described.