Mentat : An Object-Oriented Macro Data Flow System

by

Andrew S. Grimshaw
and
Jane W.S. Liu

1304 W. Springfield Avenue
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
(217)-333-0135

This work was partially supported by NASA Contract No. NAG 1-613.

Mentat : An Object-Oriented Macro Data Flow System

by

Andrew S. Grimshaw
and
Jane W.S. Liu

1304 W. Springfield Avenue
Department of Computer Science
University of Illinois
Urbana, Illinois 61801
(217)-333-0135

Abstract – Mentat, an object-oriented macro data flow system designed to facilitate parallelism in distributed systems, is presented. The macro data flow model is a model of computation similar to the data flow model with two principle differences: the computational complexity of the actors is much greater than in traditional data flow systems, and there are persistent actors that maintain state information between executions. Mentat is a system that combines the object-oriented programming paradigm and the macro data flow model of computation. Mentat programs use a dynamic structure called a future list to represent the future of computations.

This work was partially supported by NASA Contract No. NAG 1-613.

1. Introduction

In recent years, distributed systems containing interconnected workstations and host computers have been used to provide a broad spectrum of services in communication, computing, and information management and distribution. Efforts to increase parallelism and performance on distributed systems have concentrated on load balancing via remote procedure call and process migration. A remote procedure call (**RPC**) is the invocation of a procedure on a remote host. How RPC is implemented differs from system to system, but the idea is essentially the same: the invoker passes control to a remote procedure and resumes computation when the remote procedure has completed. Examples of systems that support some form of RPC include the V-System [1-3], LOCUS [4] and Accent [5]. Process migration is concerned with migrating processes from one host to another. There are two main reasons for migrating a process: the current host is heavily loaded and the destination host is lightly loaded; or it is known that the current host will soon fail. DEMOS/MP [6], ACCENT, and the V-System are examples of systems that allow processes to migrate. General process migration issues are discussed in [7] and [8]. It should be noted that RPC and process migration are not mutually exclusive; rather they are two different means to the same end.

Concurrent with advances in computer systems and communications have come advances in programming languages and software engineering. One such advance is the introduction and wide-spread acceptance of the object-oriented programming paradigm. The most important element that object-oriented languages share is the ability to define abstract types which hide implementation and data structure details from the objects which use them [9-12]. By hiding the implementation details (algorithms and data structures) it is felt that more reliable and secure software can be constructed, and software re-use and maintenance can be simplified. The object-oriented abstraction has been brought to bear on many distributed systems projects, well known examples of which include Eden and Emerald at the University of Washington [13-14], Clouds at Georgia Tech [15], and Argus at MIT [16].

One approach used to increase parallelism in tightly coupled systems is data flow [17-24]. In the well-know data flow model of Dennis [24] computation is data driven. A program is a directed graph in which the nodes are computation primitives called actors. The arcs in a data flow graph represent the data dependencies between actors. Control and data tokens generated by the actors flow along the arcs from one actor to another. When there is a token on every incoming arc of an actor it may execute, consuming the input tokens and generating one or more output tokens. In traditional data flow systems the actors are low level machine primitives. The communications overhead can become quite significant [25-28]. At what point the communications sub-system becomes a bottleneck and limits the degree of parallelism depends primarily on the ratio of the communication overhead to the execution time of the computation primitives [26-28]. In order to use the data flow approach on loosely coupled systems this ratio must be much smaller than in traditional data flow systems.

In this paper we introduce Mentat, an object-oriented system that uses the macro data flow model [25,29] as the underlying model of computation. The macro data flow model is similar to the well-known data flow model with two principle differences: the computational complexity of the actors is greatly increased, and actors may retain state information. In Mentat the operations of objects are macro data flow actors. Using the macro data flow model as the underlying model of computation can both increase parallelism and decrease

the overhead of coordinating the parallelism. The object-oriented paradigm can be used to develop macro data flow programs.

The rest of this paper is divided into five parts. Section 2 briefly describes the macro data flow model. Section 3 introduces Mentat, our object-oriented system that uses the macro data flow model of computation. Section 3 discusses the correspondence between the macro data flow model and the object-oriented paradigm and the mechanisms used to describe macro data flow program graphs. Section 4 describes a virtual macro data flow machine that provides the architectural support for Mentat. We conclude with a short summary of the paper.

2. The Macro Data Flow Model

In the macro data-flow model of distributed computations, *macro actors* perform complex, high-level functions instead of individual machine instructions. For example, in a distributed relational database system, one may choose to let computation primitives be operations such as database-read, database-write, project, select, join, etc., required to process a transaction. Similarly, in a electronic message system, jobs are requests to send and receive messages. Macro actors may be edit-file, encrypt-file, send-file, write-mailbox, etc.. The important feature is that each macro actor execution requires a large number of machine instructions. (Unless stated otherwise, we use the term actors to mean macro actors in subsequent discussion.)

Some of the macro actors are *regular actors.* They are similar to actors in the data-flow model. Regular actors have the following characteristics:

(1) All actors of a given type are equivalent.

(2) Each type of actor requires a fixed number of input tokens, each of which must be of a specific type. When all required tokens are available, the actor is enabled.

(3) An actor may execute only when enabled.

(4) An actor performs some computation, generating output tokens that depend only on the input tokens.

(5) An actor need not send the same output tokens to all its recipients.

(6) A actor may have internal state during the course of a single execution but no state information is preserved from one execution to another.

Similar to a data-flow graph, a macro data-flow graph is a high-level view of a program. Nodes in this graph are actors. There is an arc from the actor v to the actor u when there is data dependency between v and u. Tokens flow along the arcs between actors carrying both data and control information. Parallel execution of a macro data-flow program can be realized by firing each actor as it becomes enabled. It is not necessary to be concerned with synchronizing the execution of individual actors within a program because all allowed orders of execution of actors vis a vis each other are specified by the arcs in the macro data-flow graph.

The macro data-flow model described so far is a straightforward extension of the data-flow model. It makes no provision for the sharing of information between programs. All information transfers occur through the use of token flow. Hence, there is no transfer of information between data-flow programs unless there are arcs connecting actors in the corresponding data-flow graphs. Arcs linking individual data flow programs effectively make the programs into a larger program. In general, it is difficult to model interprocess

communication and global serialization required by many applications using this scheme. This scheme presumes the knowledge of other programs, their relationship to each other, and the order of their executions. This knowledge is often not attainable. For example, suppose that the programs are transactions to a database. Information is transferred between transactions since they access the same database. It is necessary to serialize the executions of transactions that read and write the same data items in order to maintain database consistency. But it is not possible to construct a composite data-flow graph modeling interleaved executions of all possible transactions to the database allowed by the database manager.

To model communication between programs, we introduce the notion of *persistent data* and *persistent actors*. Persistent actors have the same characteristics as regular actors except for points (1), (4), and (6) listed above. A persistent actor maintains state information that is preserved from one firing to the next. Hence, the output tokens generated by a persistent actor for different firings are not necessarily the same for the same input tokens. Since each instance of a particular persistent actor type can have a different internal state, different instances are not identical. We note that the notion of persistent actors is similar to the concept of monitors in Concurrent Pascal [30] and objects in object-based systems [9-16]. By adding persistent actors to the macro data-flow model, the arcs between actors in a program graph no long completely specify all dependencies between all granules of computation carried out by the system. In particular, persistent actors provide us with a way to model information transfer between actors in different programs as well as within a program. We propose to use the data-flow graph of each program to completely specify data dependency between actors within each program and limit the use of persistent actors to model synchronization and serialization between different programs.

We note that an object-oriented approach to software design and implementation provides the ideal support to the design and development of macro data-flow programs. Indeed, macro actor types are functional objects. These objects can be instantiated to provide the required functionality. The arcs carrying tokens in the macro data-flow graphs are also objects. These objects are instantiated to provide a communication facility between the functional objects. Objects can be easily implemented in programming languages such as Ada, Modula, Smalltalk, and C++, that provide data abstraction facilities to encapsulate objects, hide implementation details, and allow parametrized interfaces. Combining the object-oriented paradigm and macro data flow is the subject of the next section.

In general, the process of choosing the functionality and complexity of the computation primitives, and hence actor types or functional objects, is a software system design process. The ideal choices of computation primitives are application dependent. For applications where good response and high throughput are essential, the computation primitives should be chosen to achieve near optimal trade-off between low communication overhead and a high degree of parallelism[26-27]. For applications where fault tolerance and availability are essential, computation primitives should be chosen so that they correspond to atomic actions that can be carried out reliably and supported on a highly available basis.

3. Mentat

Mentat is a system combining the object-oriented paradigm and the macro data flow model of computation. We begin by discussing the relationship between objects and actors. Next, naming of objects and actors, and the construction of future lists are developed. A future list is a structure that allows the specification of the future of a computation and has properties in common with continuations [31-32]. Future lists are used to represent Mentat program graphs. We conclude this section with several examples that illustrate the use and flexibility of future lists.

3.1. Objects, Actors, and Tokens

For the sake of discussion we consider each object to consist of four components: a name, a representation of the data stored in the object, a set of externally visible operations, and an (optional) process executing in parallel with invocations of operations of the object. The set of externally visible operations is the object's interface. An object performs operations on itself and its inputs, generating some other object as an output. The actual implementation of an operation (called a method in Smalltalk [9]) frequently makes use of other objects and operations. How the operations are performed, what other objects are used, and the internal representation of data structures are not visible to any outside object.

In distributed object-oriented systems it is useful to distinguish between contained objects and independent objects [13, 33]. A contained object is an object which is contained in the same address space as the invoking object. Many contained objects may share the same address space. Therefore to evaluate an operation on a contained object no references to other address spaces is required. An independent object on the other hand shares no address space with any other independent object. Each independent object can be located independently of all others. An independent object can contain many contained objects.

In Mentat we also distinguish between independent and contained objects. Each Mentat object has a name and a set of operations that it can perform. Mentat objects implement a set of actors. Each actor corresponds to one of the object's operations. In the simple case an object has only one operation and one actor. For example, an encryption object would have a single operation that requires two parameters, text and key. This object is straightforward to visualize. It consists of a single actor. The actor has two input arcs, one for each parameter. The actor waits until there are tokens on both input arcs. It then fires, encrypts the text, and places the results on its output arc. A more complex example is an object with many operations consisting of more than one actor. A queue object is such an object. It has five operations, full, empty, clear, enqueue, and dequeue. Corresponding to each operation is an actor that performs the desired operation.

In the examples above, none of the actors reference any other Mentat object. In general, however, the internal implementation of a Mentat object can be quite complex, involving contained objects, and references to other Mentat objects. When a Mentat object references other Mentat objects, the Mentat object is implemented as a macro data flow subgraph. Any such decomposition is hidden from the the user of the object. This hierarchical case will be discussed in detail later. We consider now the simple case where Mentat objects are simple objects, i.e., during the execution of any operation no operations of other Mentat objects are invoked.

To invoke a Mentat object operation a set of messages (tokens) are sent to the object, one for each parameter of the desired operation. When messages for all parameters have arrived for a particular operation, i.e., when all input tokens for the corresponding actor are present, the actor is enabled and the operation is executed. In the following we will use the terms *message* and *token*, as well as *actor* and *object operation*, interchangeably.

3.2. Tagging Computations and Names of Objects and Actors

Before we can discuss how complex computations are represented and constructed we must be able to name (distinguish) objects, operations, arcs, and computation instances. This section discusses how instances of actors are named. The definitions and notations developed here are used later on in the definitions of future lists and in the description of graph hierarchies.

A *(Mentat) object name* has several components. The components are object class, persistent, bound, and ID. *Object_class* specifies the function the object implements, e.g., queue, database, etc. This is similar to a class in Smalltalk. *Persistent* is a boolean field indicating whether the named object is persistent or regular. An object name may be bound or unbound to a specific instance of an object. *Bound* names contain an ID field, and unbound names do not. An unbound name points to any instance of the object class. If, when, and how a name should be bound depends on how it is to be used, and whether it is persistent or regular. *ID* is a system wide unique identifier for a specific instance of an object. The ID could also contain hints as to where the object is to be found. For example, it may be of the form (host name, local number). Not all components of an object name need always be present in the name, e.g., if ID is present then the rest are superfluous.

An *actor_name* is a tuple <object_name,operation> which specifies a particular externally visible operation of the object specified by the object_name.

An *arc_name* is a tuple <actor_name,argument#>, or, alternatively <object_name,operation,argument#>. The argument# field indicates to which of possibly several incoming arcs of the actor the token is addressed.

Example 1 In Figure 1 the object fred has an object_name whose object_class = 'fred'. Objects of the class 'fred' have three operations named op1 through op3, each of which corresponds to an actor in 'fred'. Each of the three tokens labeled A, B and C contain an arc_name as the destination. (Note that op2 is enabled.) The arc_names are:

A = <fred,op1, argument2>

B = <fred, op2, argument1>

C = <fred, op3, argument1>.

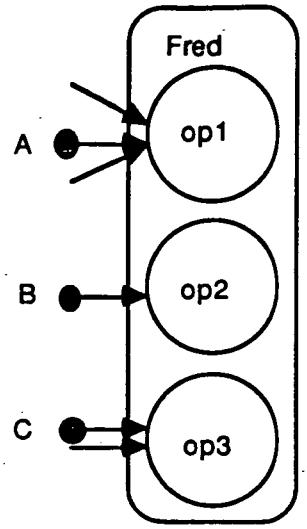

Figure 1. Arc Naming for object "fred"

Computation tags are used to match tokens belonging to the same sub-computation. A computation tag is a tuple <object_name,computation#> where the object_name is a bound name, and the computation# is a sequence number that specifies to which sub-computation of the object specified by the object_name this token belongs. The computation# of a computation tag is a unique label of a node in an elaborated data flow subgraph. Taken with the object name the computation# specifies a unique computation instance of the labeled node in the graph. Without computation tags it would be difficult to ensure that inputs for different computation instances would not be mixed together. Computation tags are similar to, and provide the same function as, token colorings in [24]. As an example, suppose there exists an encryption object with the name X. If two objects A and B wish to use the object X we must ensure that the parameters are correctly matched. Computation tags are used for this purpose. Both A and B create computation tags that are included as part of the messages transporting the parameters. The part of the messages we are interested in has the format (computation tag, data). A sends to X two messages, ((A,v) text), and ((A,v) key), where v is some integer. Similarly B sends ((B,y) text) and ((B,y) key). Now the messages can be matched on the computation tag portion to ensure that A's text is encrypted with A's key, and B's text is encrypted with B's key.

An *arc _instance* is a tuple <arc_name, computation tag> which specifies not only the computation instance, but the argument number as well. Each token must contain an arc_instance so that it may be correctly matched with other incoming tokens to fire an actor.

3.3. Program Graphs and Subgraphs

So far we have discussed objects, operations, and actors in isolation. Next we consider how actors are linked together to form program graphs and subgraphs. By allowing the dynamic substitution of subgraphs for nodes in a graph we permit the transparent implementation by subgraphs of operations that reference other Mentat objects. In this section we first illustrate the need for dynamic program graphs, and then introduce the future list, our method of representing dynamic graphs. The future list is a mechanism that facilitates the dynamic construction of program graphs and subgraphs.

In traditional data flow systems the computation is controlled by the program graph. The program graph consists of nodes and arcs. Each node, corresponding to an actor, usually contains four elements: an

opcode for the actor, space for incoming operands, a count of inputs yet to arrive, and a list of destinations where the results of the computation should be sent [22]. The program graph is usually generated by a compiler, and the nodes are then either statically allocated to processors, or allocated as they become enabled. In either case the structure of the computation is fixed at compile time, and cannot change during the course of the computation. The entire program graph is then loaded into memory and the program can be run.

Static graphs are adequate for traditional data flow programs because each actor forwards the results of its computation to a small predetermined set of actors. In the case of macro actors, determining the set of destination actors at compile time may be quite difficult, because actors can be persistent. Each instance of a persistent object must be differentiated from all others. It may not be possible to know until run time which instance of a persistent object is to be used by the program. Therefore, we cannot statically associate each node of the graph with a particular instance of the required object type. As illustrated in Example 2, it would not be possible to determine at compile time which instance of the printer server object would be used. Furthermore, enumerating all possible instances may not be feasible because the object instance actually used may not exist at compile or load time.

Example 2 Suppose we have the following code fragment.

```
Type
object printer_provider;
        { Provides a printer object with the desired characteristics. }
        get_printer(kind:string):printer_server;
        { Kind is line, laser, impact ... }
end printer_provider;
object printer_server;
        { Provides access to the printer device. }
        print(what:string);
end printer_server;
var     sally:printer_provider;
        fred:printer_server;
begin
        { Assume that sally has been bound. }
        fred:=sally.get_printer("line 132");
        { Get a printer object from the printer provider. Then }
        { print out "hello world" to the printer.                }
        fred.print("hello world");
end;
```

Our solution to this problem is to allow the dynamic construction and modification of the program graph. This is accomplished by representing program graphs as future lists. A future list is a data structure. It describes a macro data flow subgraph that is to be executed:

```
future_list == future | future future_list | epsilon | passive
future      == (arc_instance future_list)
```

Each actor receives as part of its invocation parameters a future list. The future list indicates to where the results of the computation are to be sent. If the actor wishes to invoke other actors and have them perform computations, it may augment the future list that it was passed and invoke the other actors with the new future list. When an actor completes its computation and it is time to return the results, a copy of the results are sent to each future named in it's future list. The future list is split up, and each named future in the future list receives its corresponding future list.

Each actor receives enough of the program graph to continue the computation. It is not necessary for the entire program graph to be generated at compile time. Indeed, the structure of the graph is only implied and is constantly changing as actors modify their futures. By carrying the future lists with the tokens we eliminate any need for a graph controller or static allocation. Control is completely decentralized. In addition object elaboration into subgraphs can now be completely hidden from user of the object.

Future lists can be used for other purposes than specification of the future of the computation under normal conditions. They can also be used to provide alternative paths of execution in the event of special conditions arising. For example, suppose there is a database server object that is passed several future lists. The first is used after the successful completion of the database operation. The second is to be used if the database operation fails for some reason. The third is used if a security violation is detected. Each of the futures passed represents a subgraph to be invoked by the database server on behalf of the caller when the corresponding condition is true at completion. (Invocation of a subgraph is equivalent to sending tokens to the source nodes elaborated with future lists defining the subgraph.)

3.4. Examples of Future Lists

Next we will demonstrate the use of future lists with five examples. In the following we assume that there are three objects, A, B, C. Object A consists of two actors, prelude and epilogue. Objects B and C each consist of one actor with one argument. A is the invoker (caller), B is the invoked object (callee). B is a simple object: it always forwards its results directly to the objects in its future list. We will omit the computation tag element of the arc_instance in all of the examples for clarity. We also define the following operations to simplify list construction and description:

passive	==	Returns the passive future_list.
nofuture	==	Returns an empty future list epsilon.
my_future	==	Returns the "arbitrated" future for the active invocation. The returned value is always a future_list.
my_name	==	Returns the specific object_name of the caller, similar to "self" in Smalltalk.

An actor may send a token to "my_future". This has the effect of sending a copy of the token to each actor named in the future_list, and of splitting out the future list to each of the named recipients. In this manner the results of the computation can be sent to wherever the invoker (constructor of the future list) wanted them to go. This will be the usual manner of returning from an invocation.

Example 3 Figure 2 illustrates how a particular future list corresponds to a portion of a program graph. A receives a future list of:

(<B, op1,arg1> (<D, op1,arg1> epsilon))

(<C, op1,arg1> (<D, op1,arg2> epsilon)) epsilon

A sends it's results to <B, op1,arg1> and <C, op1,arg1>. B receives a future list of (<D,op1,arg1> epsilon), and C receives a future list of (<D,op1,arg2> epsilon). B and C upon completion, in turn send the results of their computation to D. Beyond that point, the future is not specified.

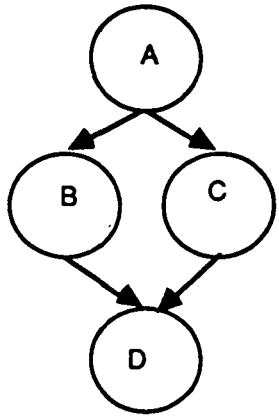

Figure 2. Correspondence of Future Lists to Graphs

Example 4 Figure 3 corresponds to a standard remote procedure call. In this scenario A calls B and waits for the result. When A is invoked it performs some computation and then sends a message to B. This is done by the prelude actor of A. Since A wishes to continue the computation with the results from B when B has completed, A waits for the return message from B. A sends a message to B with the future list (<A,epilogue,arg1> my_future). The above will cause the graph in Figure 3 to be executed. Note that this is a straight serial execution and is essentially a remote procedure call. This example is presented in order to demonstrate remote procedure call can be implemented in a data driven fashion. The serial execution of Figure 3 can also be realized by sending B the future list (<A,epilogue,arg1> passive). In this case A's current future must be saved in A, and A must ensure that the next request that it services is the epilogue operation using the results from B.

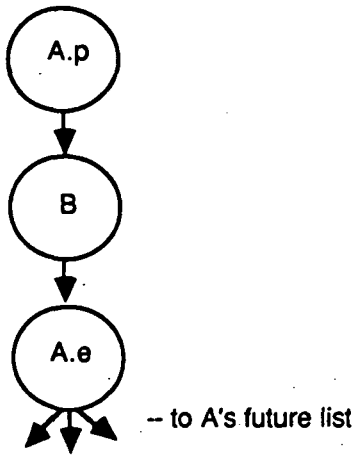

Figure 3. Remote Procedure Call

Example 5 Similarly, tail recursion, call/forward, and call/cc can be implemented in a data driven fashion. In this case the "return address" for the invocation of B from A is not A; rather A specifies that B is to complete the computation and return the results to wherever A was supposed to return them. A sends B A's own future. This can be done by passing a future of my_future. Figure 4 is the executed graph. Note how this mechanism is very similar to a tail recursive call, or a call with current continuation in Scheme. It allows A to invoke B and forget about the computation.

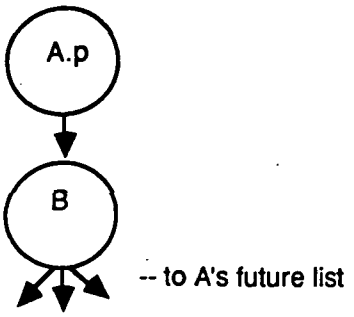

Figure 4. Continuation Passing

Example 6 Call/epsilon implies that the invoked actor need not return a result at all, or that the invoked actor is assumed to "know" what to do, i.e., the graph is hard-wired. The future is NoFuture, epsilon. Figure 5 shows this graph structure (or lack of structure!).

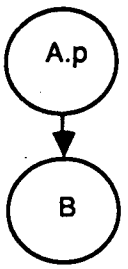

Figure 5. A "hard wired" Graph

In the preceding we assume that the operation of the object B is a simple operation: that it does not invoke any other subgraphs. If we remove that assumption, then B can behave just as A did in the previous examples, i.e., it can invoke other actors and pass it's future on to the actors it invokes. In this manner nodes of the program graph are hierarchically expanded. We will call this subgraph the elaborated subgraph of the operation. Note that the elaborated subgraph of an operation is not necessarily always the same every time the operation is executed.

The nodes of the elaborated subgraph are drawn from both other locally contained actors of the containing object, and from actors of other Mentat objects. Each node corresponds to a separate actor. Therefore, a particular actor may appear more than once in the elaborated subgraph of an operation. In order to distinguish among these possible multiple instances of a particular actor the nodes of the elaborated subgraph are uniquely labeled. These unique labels are used to construct computation tags and arc instances to aid in the matching of tokens.

Hierarchical expansions of the program graph can be accomplished in one of two ways. First, an actor can construct a future list that consists of the elaborated subgraph and the current future list. Its results are forwarded to the elaborated subgraph, and the results of the elaborated sub-graph are forwarded to its future. (Shown in Figures 4 and 7.) The second method involves the construction of appropriate future lists and sending those lists along with tokens to the appropriate nodes of the elaborated subgraph. The next example demonstrates the first of these two mechanisms. (Note that we will no longer include operation names and argument numbers in arc_names for clarity.)

Example 7 Assume that A, B, and C are as before. We add E, an object with one operation of two arguments; and D, F, G, H, which are objects like C. Suppose A is initially invoked and wishes to invoke the subgraph in Figure 6. A sends a future to B of (C (E my_future))(D (E passive)).

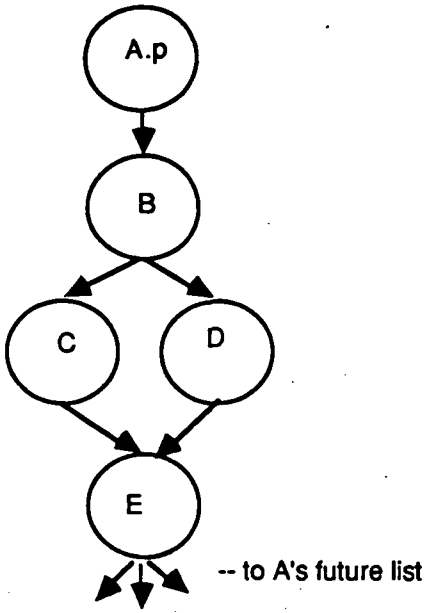

Figure 6. A's Elaborated Subgraph

If B is further elaborated into Figure 7, B modifies its future list to be my_future = (F (I my_future))(G (I passive))(H (I passive)).

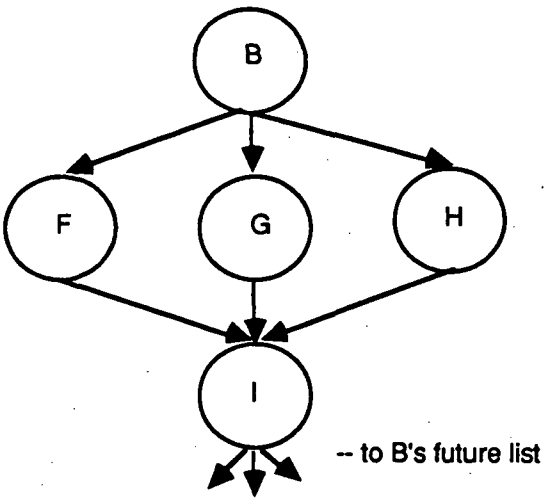

Figure 7. B's Elaborated Subgraph

This causes a total graph execution of Figure 8.

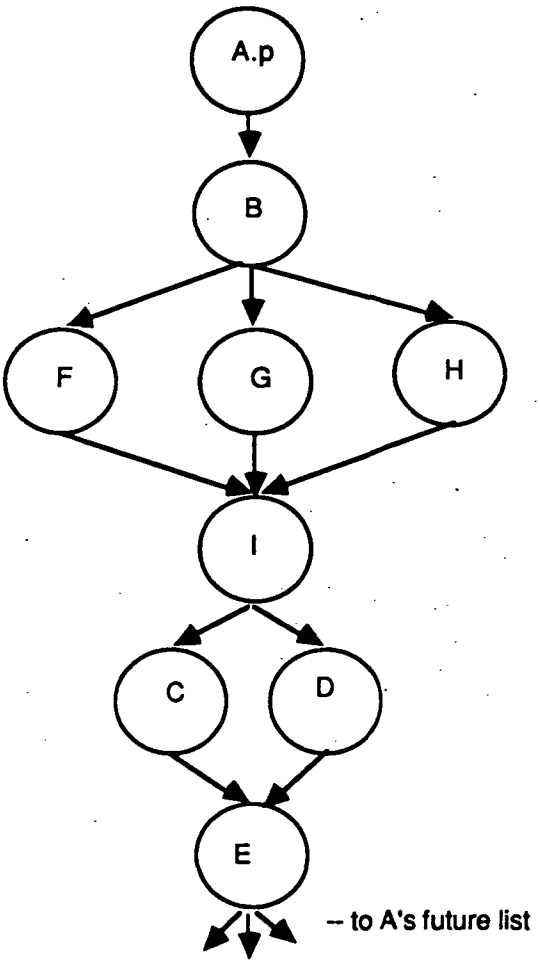

Figure 8. Total Subgraph of A's Execution

As one can see complex graph structures can be created in this manner. Future lists provide a generalized way to specify, and hierarchically augment, the macro data flow graph of a program.

3.5. Performance Issues of Future Lists

It is important that future lists can be implemented efficiently. The amount of data passed between actors can be reduced in three ways. First, consider the use of passive future lists. When an actor requires two or more tokens to fire, all but one of the received future lists are redundant. We have provided passive future lists to reduce the communication overhead. The use of a passive future list in a message indicates that some other argument to the actor carries the future list for the actor computation. (The use of passive future lists is illustrated in Example 7.) Passive future lists are usually quite short.

Second, in many instances the current future list of an object need not be passed to the elaborated subgraph. Instead the object retains its current future, starts the subgraph, and waits for the results. When the subgraph completes the object forwards the results to its future. This is illustrated by the second future in Example 4. In that example, only the future lists for the elaborated subgraph of the object need to be transported.

Third, consider a segment-oriented architecture that supports variable sized segments. Each future list or message could be contained in a separate segment. To move a future list or message from one object's address space to another object's address space on the same machine, only the segment descriptor tables need to be modified. Similarly, it is possible that pointer passing could be employed if

sufficient memory protection were available.

4. The Virtual Macro Data Flow Machine

A virtual macro data flow machine that executes macro data flow programs is shown below in Figure 9. Each processor in a distributed system may execute one or more virtual macro data flow machines. Alternatively, a cluster of processors may jointly support one virtual macro data flow machine. The function of each component is explained in more detail in the following paragraphs. We note again that each object, stored in the object storage unit, may consist of several actors; and that each actor is a computation resource of arbitrary complexity. Methods to efficiently implement the virtual macro data flow machine on different hardware architectures will be discussed in a later paper.

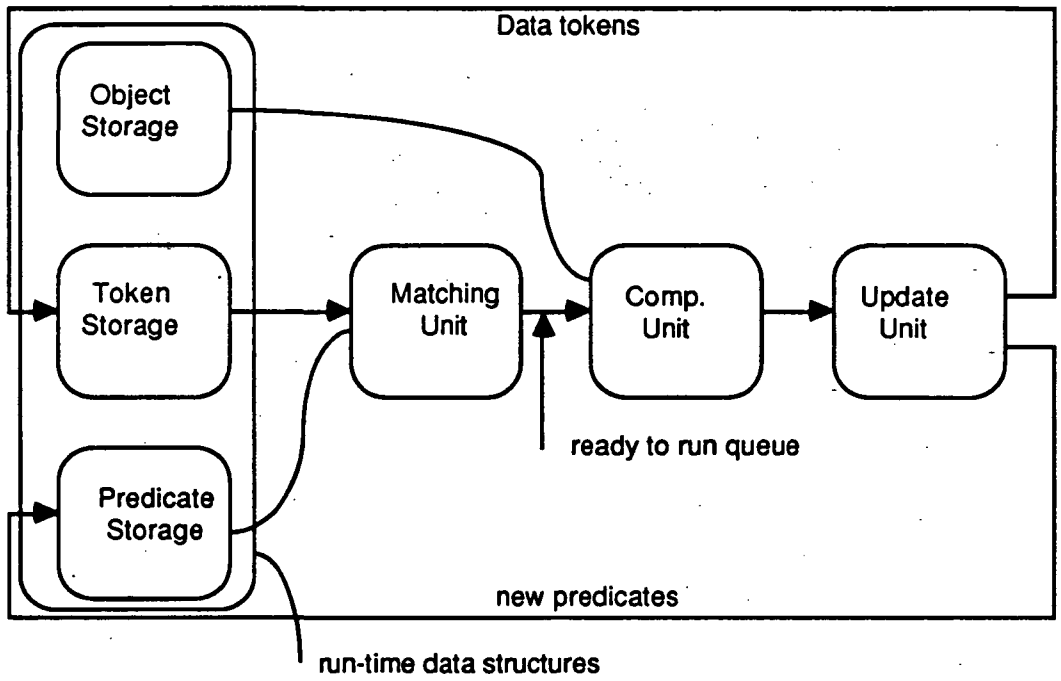

Figure 9. Virtual Macro Data Flow Machine

4.1. The Matching Unit

The function of the matching unit is twofold; to determine which actors are ready to fire and to ensure that once actors are enabled, they are invoked and their tokens are delivered to them. Each object has associated with it a current predicate and a message queue. The predicate specifies which actors are available for firing. The predicate is a collection of boolean expressions, one for each actor. The value of each expression depends on the contents of the message queue and the value of objects contained in the object. The simplest predicates require only that a set of matching messages have arrived on the incoming arcs of an actor.

When a new message arrives for an actor, the matching unit determines if that message enables the actor. To determine whether an actor is enabled by the receipt of a message, the matching unit applies the object's predicate to the messages stored in the token storage unit. Three examples illustrate the use of predicates.

Example 8 Consider the following encryption object (in C++) used earlier.

```
class encrypt {                    // A DES encryption object
public
        char*    crypt(char* , char* );
}
char* encrypt::crypt(char* key, char* text)
{
                code for DES encryption
}
```

The predicate for this object only requires that the arguments have the same computation tag. The predicate is ((operation=crypt) and (argument1.computation_tag=argument2.computation_tag)).

Example 9 This example (an ADA queue object) demonstrates how guards are included into the predicate.

```
task type queue is
        entry full() return BOOLEAN;
        entry empty() return BOOLEAN;
        entry clear() return BOOLEAN;
        entry enq(val: in INTEGER);
        entry deq(val: out INTEGER);
end queue;
task body queue is
        qmax:constant:=10;
        subtype qindex is INTEGER range 0..(qmax-1);
        numq, head, tail:qindex:=0;
        q:array(qindex) of integer;
begin
   loop
        select
                when (numq< qmax) =>
                accept enq(val:in INTEGER) do
                        qhead]:=val;
                        head:=(head + 1) mod qmax;
                        numq:=numq + 1;
                end enq;
        or      when (numq > 0) =>
                accept deq(val:out INTEGER) do
                        val:=q[tail];
                        tail:=(tail + 1) mod qmax;
                        numq:=numq-1;
                end deq;
        or      accept full() do
                        if numq=qmax-1 then return(TRUE)
                        else return(FALSE);
                end full;
        or      accept empty() do
                        if numq=0 then return(TRUE)
                        else return(FALSE);
                end empty;
        or      accept clear() do
                        head:=0;
                        tail:=0;
                        numq:=0;
                        return(TRUE);
                end clear;
        end select;
   end loop;
end queue;
```

In this example only local variables and constants of the object are used. Each accept statement corresponds to an actor in the queue object. Because all operations of the queue object require at most one argument there is no need to match on the computation tag field of the message. The predicate is:

((operation = enq) and (numq < qmax))

or ((operation = deq) and (numq > 0))

or (operation = full)

or (operation = empty)

or (operation = clear).

Example 10 Example 10 illustrates the use of both local variables and the values contained in the messages in guards. Note that in this example we stretch the definition of ADA slightly.

```
task type account is
        entry withdrawal(w_amount:in INTEGER) return BOOLEAN;
        entry deposit(amount:in INTEGER) return BOOLEAN;
end account;
task body account is
        balance:INTEGER:=0;
begin
   loop
        select
                when (balance - w_amount > 0) =>
                accept withdrawal(w_amount: in INTEGER) do
                        balance:=balance - w_amount;
                        return(TRUE);
                end withdrawal;
        or      accept deposit(amount:in INTEGER) do
                        balance:=balance + amount;
                        return(TRUE);
                end deposit;
        end select;
   end loop;
end account;
```

In this example the predicate references both a local object, the balance field, and a field of the message, the w_amount field. The predicate is:

((operation = withdrawal) and (balance - w_amount > 0))

or (operation = deposit).

The process of determining if an arriving message satisfies a predicate falls into three cases. The first two cases are straightforward to implement; however the third case is more complex. First, if the actor name is a bound name then the message is addressed to a specific actor. All arguments for that actor will arrive at the same instance of the actor. Since all arguments will arrive eventually, there is no need for the matching unit to look elsewhere for the other arguments. Second, if the actor name is unbound and the actor requires only one argument then the matching unit can enable an instance of the actor. Since all instances are the same, there is no need to use a particular instance. Third, if the name is unbound and the actor requires more than one argument, then the matching unit or the token storage unit must determine if the other arguments have already been generated and are waiting at another token storage unit. This requires cooperation among the different matching units in the system.

Once the matching unit has determined that an actor is enabled by the arrival of a message, it places that actor in the ready-to-run queue, and marks the object predicate as invalid to ensure that no two actors of the same object instance are ever in the ready queue at the same time. Two types of errors could result if this condition is not maintained. The first can be illustrated using the account object above. Suppose that the balance is $200 and two withdrawal $150 messages are sent to the same instance of the object. Clearly if both arrive before either operation is performed both would satisfy the predicate. But if both operations are performed then the balance would be less than $0, a condition that is not supposed to occur. The second type of error could occur if more than one actor at a time of the same object instance is allowed to execute on a processor. Since all actors in the same object share the same address space, traditional synchronization problems would result.

4.2. Token Storage Unit

The token storage unit is responsible for ensuring that tokens with specific addresses are delivered to the correct actor. The token storage unit and the matching unit work closely together. When the token storage unit receives a token destined for an actor on the machine on which the token storage unit resides, the token storage unit notifies the matching unit that a token has arrived for the named actor. The matching unit then determines whether the new token enables the actor. The token storage unit can be thought of as a database. The database may be queried, read, or have insertions performed on it. The records of the database are the individual messages. Furthermore, the database notifies the matching unit when insertions are made.

4.3. Predicate Storage Unit

The predicate storage unit is an object which stores the predicates on which currently instantiated objects are waiting and the predicates of all available functional objects. The matching unit applies the predicates stored in the predicate storage unit to messages stored in the token storage unit to determine if the arrival of a message has enabled an actor to fire. The predicate storage unit is updated by the matching unit when it fires an actor, disabling the predicate of the associated object; and by the update unit when an active computation blocks on predicate.

4.4. Update Unit

The update unit is responsible for forwarding the results of the computation to the actors named in the future list and updating the predicate storage unit with a new predicate. Forwarding the results of the computation to the future of the computation consists of two steps. First, the individual futures are extracted from the future list of the actor. Then, for each future extracted a message is sent to the arc_instance of that future. Each message contains the arc_instance and future list of the extracted future, as well as the results of the computation. The message format is:

(arc_instance, future list, computation result).

Example 11 In Figure 2, actor A receives a future of:

(<B, op1,arg1, computation tag1> (<D, op1,arg1, computation tag3> epsilon))

(<C, op1,arg1, computation tag2> (<D, op1,arg2, computation tag3> epsilon)).

Suppose that the result of the actor A computation is "value". The update unit passes to the token storage unit the two messages:

(<B, op1,arg1, computation tag1>, (<D, op1,arg1, computation tag3> epsilon), value)

and (<C, op1,arg1, computation tag2>, (<D, op1,arg2, computation tag3> epsilon), value).

The token storage unit is then responsible for transporting the messages to the correct objects.

4.5. Computation Unit

The computation unit is responsible for executing actors that have been enabled and placed on the ready-to-run queue. It examines the ready-to-run queue and selects an actor to execute. It receives the address of the actor and the data tokens the actor requires from the ready-to-run queue. The computation unit then executes the actor until the actor completes. When the actor completes, the computation unit notifies the update unit, and then begins again by examining the ready-to-run queue. Note that because we know that no two actor computations of different object instances can interfere with one another the computation unit may execute more than one actor at a time. However, the total throughput of the machine will be reduced due to the overhead of task switching between actors if more than one actor is executed at a time.

5. Summary

In this paper we have discussed an object-oriented macro data flow system, Mentat. The macro data flow model is a model of computation similar to the data flow model with two principle differences: the computational complexity of the actors is much greater than in traditional data flow systems, and, there are persistent actors that maintain state information between executions. Mentat is a system that combines the object-oriented programming paradigm and the macro data flow model of computation. Mentat programs are represented by future lists. A future list is a dynamic structure that represents the future of the computation; it is similar to a continuation. Because the macro data flow model includes persistent actors, future lists must be dynamic. Several examples illustrating the use of future lists were presented. A virtual macro data flow machine was presented. The machine provides the architectural support needed to execute Mentat programs. Each host in a distributed system executes one or more copies of the machine.

References

[1] Theimer, Marvin M., Lantz, Keith A., and David R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings of the Tenth ACM Symposium on Operating System Principles*, December, 1985.

[2] Cheriton, David R., and Willy Waenepoel, "Distributed Process Groups in the V Kernel," Stanford University.

[3] Cheriton, David R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, pp. 19-43, vol. 1, no. 2, 1984.

[4] Popek, F. et al., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the 8th Symposium on Operating System Principles*, pp.169-177, Dec. 1981.

[5] Rashid, R. F., and G. G. Robertson, "Accent: A communication oriented network oriented operating system kernel," *Proceedings of the 8th Symposium on Operating System Principles*, pp.64-75, Dec. 1981.

[6] Powell, Michael L., and Barton P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pp.110-119, 1983.

[7] Eager, D. L., Lazowska, E. D., and J. zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, pp.662-675, Vol. SE-12, No. 5, May 1986.

[8] Svobodova, Liba,"Resilient Distributed Computing,"*IEEE Transactions on Software Engineering*, vol. SE-10, no. 3, pp. 257-267, May, 1984.

[9] Goldberg, A., Robson D., "Smalltalk-80: The Language and its Implementation," Addison-Wesley, Reading MA, 1983.

[10] "Reference Manual for the Ada Programming Language," United States Department of Defense, Ada Joint Program Office, July 1982.

[11] Wirth, N., "Programming in Modula-2," Springer-Verlag, Berlin, 1983.

[12] Stroustrup, Bjarne,"The C++ Programming Language," Addison-Wesley, Reading MA, 1986.

[13] Black, A., Hutchinson, N., Jul E., and Henry Levy, "Distribution and Abstract Types in Emerald," Univeristy of Washington, TR 85-08-05, August, 1985.

[14] Lazowska, E. D., et. al.,"The Architecture of the Eden System," *Proceedings of the 8th Symposium on Operating System Principles*, ACM, pp. 148-159, December, 1981.

[15] LeBlanc, Richard H., and C. Thomas Wilkes,"Systems Programming with Objects and Actions," *Proceedings 5th Distributed Computer Systems*, IEEE 1985, pp. 132-139.

[16] Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, pp. 381-414, 1983.

[17] Srini, Vason P.,"An Architectural Comparison of Dataflow Systems," *IEEE Computer*, pp. 68-88, March, 1986.

[18] Gajski, D.D., Padua, D.A., Kuhn, R.H., and D. J. Kuck,"A Second Opinion on Data Flow Machines and Languages,"*IEEE Computer*, vol. 15, no. 2, pp. 58-69, February, 1982.

[19] Ackerman, William B.,"Data Flow Languages,"*IEEE Computer*, vol. 15, no. 2, pp. 15-25, February, 1982.

[20] Treleaven, Philp C., Brownbridge, David R., and Richard P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture,"*Computing Surveys*, vol. 14, no. 1, pp. 93-143, March, 1982.

[21] Agerwala, T., and Arvind,"Data Flow Systems," *IEEE Computer*, vol. 15, no. 2, pp. 10-13, February, 1982.

[22] Johnson, Douglas, et. al.,"Automatic Partitioning of Programs in Multiprocessor Systems,"*Proceedings of the 7th Symposium on Operating System Principles*, IEEE, pp. 175-178, 1980.

[23] Vedder, R., Campbell, M., and G. Tucker,"The Hughs Data Flow Multiprocessor,"*Proceedings of the 5th Conference on Distributed Computing Systems*, IEEE, pp. 2-9, 1985.

[24] Dennis, J., "First Version of a Data Flow Procedure Language, "MIT TR-673, May, 1975.

[25] Liu, Jane W. S., and Andrew Grimshaw, "A Distributed System Architecture Based on Macro Data Flow Model," *Proceedings Workshop on Future Directions in Architecture and Software*, (sponsored by the Army Research Office,) South Carolina, May 7-9, 1986.

[26] Gaudiot, J. L., and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proceedings of the 1984 IEEE Conference on Distributed Systems*, 1984.

[27] Brock, J. D., Omondi, A. R., and David A. Plaisted, "A Multiprocessor Architecture for Medium-Grain Parallelism," *Proceedings of the 6th International Conference on Distributed Systems*, pp.167-174, May 1986.

[28] Babb, Robert F.,"Parallel Processing with Large-Grain Data Flow Techniques,",*IEEE Computer*, pp. 55-61, July, 1984.

[29] Liu, Jane W. S., and Andrew Grimshaw, "An object-oriented macro data-flow architecture,"*Proceedings of the 1986 National Communications Forum*, September, 1986.

[30] Hansen, Per Brinch,"The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, pp. 199-207, vol. SE-1, no. 2, June, 1975.

[31] Abelson, Harold, Sussman, Gerald Jay, with Julie Sussman,"Structure and Interpretation of Computer Programs," The MIT Press,Cambridge Mass.,1985.

[32] Stoy, Joseph E.,"Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," The MIT Press,Cambridge Mass., 1977.

[33] Nierstrasz, O.M., "Hybrid: A Unified Object-Oriented System," *IEEE Database Engineering*, vol. 8, no. 4, pp. 49-57, December, 1985.