Mentat 2.5 Programming Language Reference Manual

The Mentat Research Group

Technical Report No. CS-94-05
February 17, 1994

Mentat 2.5 Programming Language Reference Manual

The Mentat Research Group

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

This work is funded in part by NSF grants ASC-9201822 and CDA-8922545-01, and NASA grant NGT-50970.

The following people have contributed to the Mentat project: Andrew Grimshaw, Ed Loyot, Jon Weissman, Padmini Narayan, Emily West, John Karpovich, Laurie MacCallum, Tim Strayer, Brian Paine, David Mack, Virginio Vivas, and Gorell Cheek.

Table of Contents

1.0 Introduction 2
2.0 The Mentat Programming Language 3
3.0 Mentat Classes 6
4.0 The Create() and Bind() Member Functions 9
5.0 The Destroy() Member Function 10
6.0 The Return-to-Future rtf() Mechanism 10
7.0 Select/Accept 12
8.0 Parameter Passing 14
9.0 Restrictions 14
10.0 Warnings 16
11.0 Extended C++ Examples 17
12.0 Summary 17
13.0 Programming Language Points 18
14.0 References 19

Mentat 2.5 Programming Language Reference Manual

A programmer's reference to the Mentat object-oriented parallel processing environment

1.0 Introduction

One problem facing the designers of parallel and distributed systems is how to simplify the writing of programs for these systems. Proposals range from automatic program transformation systems that extract parallelism from sequential programs, to the use of side-effect free languages, to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism. The problem with fully automatic schemes is that they are best suited for detecting small grain parallelism. The problem with schemes in which the programmer is completely responsible for managing the parallel environment is that complexity can overwhelm the programmer. Mentat strikes a balance between fully automatic and fully explicit schemes.

There are two primary components of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system. MPL is an object-oriented programming language based on C++ [Stroustrup] that masks the difficulty of the parallel environment from the programmer. The granule of computation is the Mentat class instance, which consists of contained objects (local and member variables), their procedures, and a thread of control. Programmers are responsible for identifying those object classes that are of sufficient computational complexity to allow efficient parallel execution. Instances of Mentat classes are used just like ordinary C++ classes, freeing the programmer to concentrate on the algorithm, not on managing the environment. The data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention. By splitting the responsibility between the compiler and the programmer we exploit the strengths of each, and avoid their weaknesses. Our underlying assumption is that the programmer can make better granularity and partitioning decisions, while the compiler can correctly manage synchronization. This simplifies the task of writing parallel programs, making the power of parallel and distributed systems more accessible.

This manual describes the MPL. We assume that the reader is familiar with the Mentat approach to parallel processing, and with the C++ programming language. The manual is designed to be used in conjunction with the Mentat system distribution that includes an examples directory. The examples in the directory are complete running programs and can be used as templates when building your first Mentat applications. We recommend that you attempt some simple applications with Mentat before plunging into your application. This will give you experience using the language and the run-time system tools. In this document we will illustrate important points using code fragments as opposed to complete programs. The remainder of this manual is in seven sections. Sections 2 through 6 introduce the language and describe the language features. Section 7 discusses restrictions, and Section 8 briefly describes the examples.

2.0 The Mentat Programming Language

MPL is an extended C++ designed to simplify the task of writing parallel applications by providing parallelism encapsulation. Parallelism encapsulation takes two forms, *intra-object* encapsulation and *inter-object* encapsulation. In intra-object encapsulation of parallelism, callers of a Mentat object member function are unaware of whether the implementation of the member function is sequential or parallel, i.e., whether its program graph is a single node or a parallel graph. In inter-object encapsulation of parallelism, programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke.

The basic idea in the MPL is to allow the programmer to specify those C++ classes that are of sufficient computational complexity to warrant parallel execution. This is accomplished using the **mentat** keyword in the class definition. Instances of Mentat classes are called Mentat objects. The programmer uses instances of Mentat classes much as he would any other C++ class instance[1]. The compiler generates code to construct and execute data dependency graphs in which the nodes are Mentat object member function invocations, and the arcs are the data dependencies found in the program. Thus, we generate inter-object parallelism encapsulation in a manner largely transparent to the programmer. All of the communication and synchronization is managed by the compiler.

Of course any one of the nodes in a generated program graph may itself be transparently implemented in a similar manner by a subgraph. Thus we obtain intra-object parallelism encapsulation; the caller only sees the member function invocation.

1. The differences are described in Section 9.0, Restrictions. The primary difference is that parameter passing is by-value.

MPL is built around four principle extensions to the C++ language. The extensions are Mentat classes, Mentat object instantiation, the return-to-future mechanism, and guarded select/accept statements.

The Mentat Philosophy on Parallel Computing

The Mentat philosophy on parallel computing is guided by two observations. First, that the programmer understands the problem domain of the application and can make better data and computation partitioning decisions than can compilers. The truth of this is evidenced by the fact that most successful production parallel applications have been hand-coded using low-level primitives. In these applications the programmer has decomposed and distributed both the data and the computation. Second, the management of tens to thousands of asynchronous tasks, where timing dependent errors are easy to make, is beyond the capacity of most programmers unless a tremendous amount of effort is expended. The truth of this is evidenced by the fact that writing parallel applications is almost universally acknowledged to be far more difficult than writing sequential applications. Compilers, on the other hand, are very good at ensuring that events happen in the right order, and can more readily and correctly manage communication and synchronization, particularly in highly asynchronous, non-SPMD, environments.

Intra-Object and Inter-Object Parallelism Encapsulation

A key feature of Mentat is the transparent encapsulation of parallelism within and between Mentat object member function invocations. Consider for example an instance `matrix_ops` of a `matrix_operator` Mentat class with the member function `mpy` that multiplies two matrices together and returns a matrix. As a user, when I invoke `mpy` in `X = matrix_op.mpy(B,C);` it is irrelevant whether mpy is implemented sequentially or in parallel; all I care about is whether the correct answer is computed. We call the hiding of whether a member function implementation is sequential or parallel intra-object parallelism encapsulation.

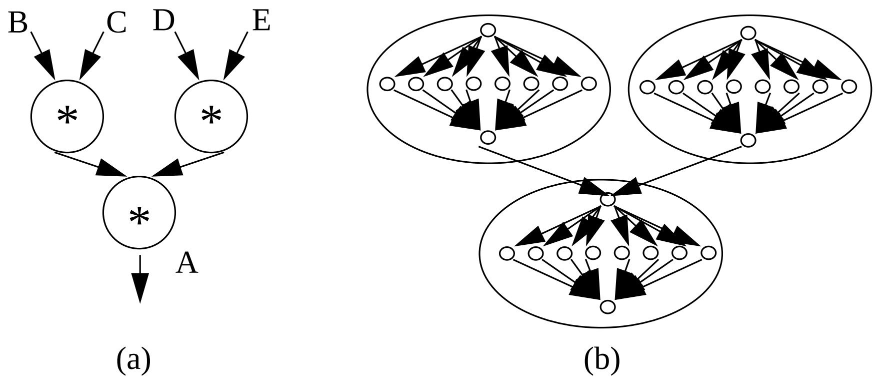
Similarly we make the exploitation of parallelism opportunities *between* Mentat object member function invocations transparent to the programmer. We call this inter-object parallelism encapsulation. It is the responsibility of the compiler to ensure that data dependencies between invocations are satisfied, and that communication and synchronization are handled correctly.

Intra-object parallelism encapsulation and inter-object parallelism encapsulation can be combined. Indeed, inter-object parallelism encapsulation within a member function implementation is intra-object parallelism encapsulation as far as the caller of that member function is concerned. Thus, multiple levels of parallelism encapsulation are possible, each level hidden from the level above.

To illustrate parallelism encapsulation, suppose `X, A, B, C, D` and `E` are `matrix` pointers. Consider the sequence of statements

X = matrix_op.mpy(B,C);
A = matrix_op.mpy(X,matrix_op.mpy(D,E));

Figure 1 **Parallel Execution of Matrix Multiply Operations**



(a) Inter-object parallelism encapsulation.

(b) Intra-object parallelism encapsulation where the multiplies of (a) have been transparently expanded into parallel subgraphs.

On a sequential machine the matrices B and C are multiplied first, with the result stored in X, followed by the multiplication of D and E. The final step is to multiply X by the result of D*E. If we assume that each multiplication takes one time unit, then three time units are required to complete the computation.

In Mentat, the compiler and run-time system detect that the first two multiplies, B*C and D*E, are not data dependent on one another and can be safely executed in parallel, as shown in Figure 4*a*. The two matrix multiplications will be executed in parallel, with the result automatically forwarded to the final multiplication. That result will be forwarded to the caller, and associated with A.

The difference between the programmer's sequential model, and the parallel execution of the two multiplies afforded by Mentat, is an example of inter-object parallelism encapsulation. In the absence of other parallelism, or overhead, the speedup for this example is a modest 1.5:

$$\text{Speedup} = \frac{T_{\text{Sequential}}}{T_{\text{Parallel}}} = \frac{3}{2} = 1.5 \quad \text{(EQ 1)}$$

However, that is not the end of the story. Additional, intra-object, parallelism may be realized within the matrix multiply. Suppose the matrix multiplies are themselves executed in parallel (with the parallelism detected in a manner similar to the above). Further, suppose that each multiply is executed in eight pieces (Figure 4*b*). Then, assuming zero overhead, the total execution time is 0.125 + 0.125 = 0.25 time units, resulting in a speedup of 3/0.25= 12. As matrix multiply is implemented using more pieces, even larger speedups result. The key point is that the programmer need not be concerned with data dependence detection, communication, synchronization, or scheduling; the compiler does it.

3.0 Mentat Classes

In C++, objects are defined by their class. Each class has an interface section in which member variables and member functions are defined. Not all class objects should be Mentat objects. In particular, objects that do not have a sufficiently high communication ratio, i.e., whose object operations are not sufficiently computationally complex, should not be Mentat objects. Exactly what is complex enough is architecture dependent. In general, several hundred instructions long is a minimum. At smaller grain sizes the communication and run-time overhead takes longer than the member function, resulting in a slow-down rather than a speed-up.

Mentat uses an object model that distinguishes between two “types” of objects, contained objects and independent objects.[2] Contained objects are objects contained in another object’s address space. Instances of C++ classes, integers, structs, and so on, are contained objects. Independent objects possess a distinct address space, a system-wide unique name, and a thread of control. Communication between independent objects is accomplished via member function invocation. Independent objects are analogous to UNIX processes. Mentat objects are independent objects.

Because Mentat objects are address space disjoint, member function calls are call by value. Results of member functions are also returned by value. Pointers to objects, particularly variable size objects, may be used as both parameters and as return types. However, a copy of the object pointed to is made and transmitted. To provide the programmer a way to control the degree of parallelism, Mentat allows both standard C++ classes and Mentat classes. By default, a standard C++ class definition defines a standard C++ object.

The programmer defines a Mentat class by using the keyword **mentat** in the class definition (see Figure 4). The programmer may further specify whether the class is **persistent**, **sequential**, or **regular**. The syntax for Mentat class definitions is:

new_class_def::	mentat_definition class_definition |
	class_definition
mentat_definition::	**persistent mentat** |
	sequential mentat |
	regular mentat
class_definition::	**class** class_name {class_interface};

Persistent and sequential objects maintain state information between member function invocations, while regular objects do not. Thus, regular object member functions are pure functions. Because they are pure functions the system is free to instantiate new instances of regular classes at will. Regular classes may have local variables much as

2. The distinction between independent and contained objects is not unusual, and is driven by efficiency considerations.

procedures do, and may maintain state information for the duration of a function invocation.

When should a class be a Mentat class? In three cases: when its member functions are computationally expensive, when its member functions exhibit high latency (e.g., IO), and when it holds state information that needs to be shared by many other objects (e.g., shared queues, databases, physical devices). Classes whose member functions have a high computation cost or high latency should be Mentat classes because we want to be able to overlap the computation with other computations and latencies, i.e., execute them in parallel with other functions. Shared state objects should be Mentat classes for two reasons. First, since there is no shared memory in our model, shared state can only be realized using a Mentat object with which other objects can communicate. Second, because Mentat objects service a single member function at a time, they provide a monitor-like synchronization, providing synchronized access to their state.

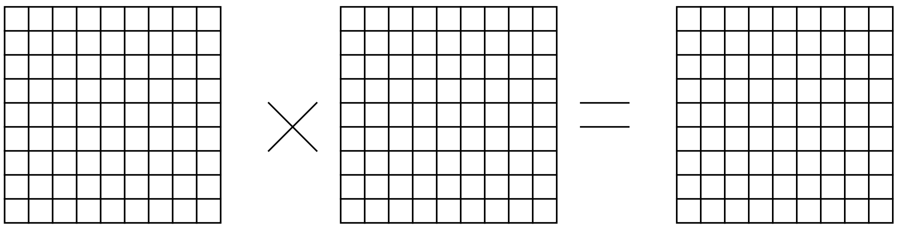
To illustrate the difference between regular and persistent mentat classes, suppose we wish to perform matrix operations in parallel, e.g., a matrix-matrix multiply. Recall that in a matrix-matrix multiply a new matrix is formed. Each element in the result is found by performing a dot product on the appropriate rows and columns of the input matrices (Figure 4*a*). Because matrix-matrix multiply is a pure function, we could choose to define a regular mentat class matrix_operators as in Figure 4*b*. In this case, every time we invoke a mpy() a new mentat object is created to perform the multiplication and the arguments are transported to the new instance. Successive calls result in new objects being created and the arguments being transported to them.

Alternatively, we could choose to define a persistent mentat class `p_matrix` as in Figure 4*c*. To use a `p_matrix`, an instance must first be created and initialized with a `matrix*`. Matrix-matrix multiplication can now be accomplished by calling `mpy()`. When `mpy()` is used the argument matrix is transported to the already existing object. Successive calls result in the argument matrices being transported to the same object. In both the persistent and regular case the implementation of the class may hierarchically decompose the object into sub-objects, and operations into parallel sub-operations. This is an example of intra-object parallelism encapsulation.

A sequential Mentat object is a special type of persistent Mentat object. A sequential Mentat object allows the user to control the order of invocation of member functions. Recall that for Mentat objects, member functions are invoked as soon as all of their arguments are available, irrespective of their order within the caller. However, there are circumstances which may require a specific ordering of the member function calls. These semantics are possible with a persistent Mentat object if the caller blocks after each member function invocation, however, this approach limits the amount of concurrency that can be achieved by the application. A sequential persistent Mentat object will enforce these semantics automatically, the caller need not block after each invocation.

An instance of a Mentat class is a *Mentat object*. All Mentat objects have a separate address space, a thread of control, and a system-wide unique name. Instantiation of Mentat objects is slightly different from standard **C++** object instantiation semantics. First, consider the **C++** fragment:

Figure 2 **Regular versus Persistent Classes to Perform Matrix Multiplication**



(a) matrix-matrix multiplication

```
regular mentat class matrix_operators {
// private members
public:
   matrix* mpy(matrix*,matrix*);
};
```

(b) regular mentat class definition

```
persistent mentat class p_matrix {
// private members
public:
   void initialize(matrix*);
   matrix* mpy(matrix*);
};
```

(c) persistent mentat class definition

```
{// A new scope
        int X;
        p_matrix mat1;
        matrix_operators m_ops;
} // end of scope
```

In C++, when the scope in which `X` is declared is entered, a new `integer` is created on the stack. In the MPL, because `p_matrix` is a Mentat class, `mat1` is a *name* of a Mentat object of type `p_matrix`. It is not the instance itself. Thus, `mat1` is analogous to a pointer. Names are also known as *Mentat variables*.

Mentat variables (e.g., `mat1`) can be in one of two states, *bound* or *unbound*. An unbound name refers to any instance of the appropriate Mentat class. A bound name refers to a specific instance with a unique name. When a Mentat variable comes into scope or is allocated on the heap, it is initially an unbound name: it does not refer to any particular instance of the class. Thus, a new `p_matrix` is not instantiated when `mat1` comes into scope. When unbound names are used for regular Mentat classes (e.g., `m_ops`), the underlying system logically *creates a new instance for each invocation* of a member function. This can lead to high levels of parallelism.

4.0 The `Create()` and `Bind()` Member Functions

The programmer binds Mentat variables to persistent Mentat objects using two new reserved member functions for all Mentat class objects: `create()` and `bind()`. There are three ways a Mentat variable (e.g., `mat1`) may become bound: it may be explicitly created using `create()`, it may be bound by the system to an existing instance using `bind()`, or the name may be assigned to a bound name by an assignment. The `bound()` function indicates whether the mentat object is bound to a particular instance. The member function `destroy`() destroys the named persistent Mentat object. If the name is unbound, the call is ignored.

The `create()` call tells the system to instantiate a new instance of the appropriate class. There are five flavors of `create()`. Assume the definition `p_matrix mat1:`

1. mat1.create();
2. mat1.create(another_M_object); // colocate the object
3. mat1.create(int on_host); // place on a specific host
4. mat1 = expression;
5. mat1.bind(int scope);

When `create()` is used as in (1), the system will choose on which processor to instantiate the object. The programmer may optionally provide location hints. These hints allow the programmer to specify where he wants the new object to be instantiated. In (2), the programmer has specified that the new Mentat object should be placed on the same processor as the Mentat object `another_M_object`. In (3), the programmer has specified that the new object should be placed on a specific processor, where `on_host` is the processor number (the processor number is assigned in order of appearance in the config file, starting with 0). Names may also be bound as the result of assignment to an expression, as in (4).

The `create()` function may be overloaded by the programmer in order to pass arguments into the object in a manner similar to that used in constructors. When create is overloaded the overloaded parameters may not conflict with the above options. This will be fixed in a later release. The effect of an overloaded create call is to first create the object, and then invoke the overloaded member function on the object. Note that unlike other Mentat object member function invocations, this call is blocking, i.e., the calling thread does not proceed until the create call has completed successfully. The overloaded create member function must return void, and must `rtf()`.

Mentat variables may also be bound to an already existing instance using the `bind(-int scope)` member function, as shown in (5). The integer parameter `scope` can take one of three values: `SEARCH_LOCAL`, `SEARCH_SUBNET`, and `SEARCH_GLO-BAL`. This restricts the search for an instance to the local host, the local subnet, or the entire system, respectively.

5.0 The `Destroy()` Member Function

The member function `destroy()` destroys the named persistent Mentat object. If the name is unbound, the call is ignored. Once destroyed, a Mentat variable cannot be reused. This is an implementation restriction that we expect to relax in the future. Care must be taken when using `destroy()`. If the name is in use by more than one Mentat object the “dangling pointers” problem occurs as when using pointers and the heap. An additional complication is that you may destroy the object before all operations applied to the object have completed.

6.0 The Return-to-Future `rtf()` Mechanism

The return-to-future function (`rtf()`) is the Mentat analog to the `return` of **C**. Its purpose is to allow Mentat member functions to return a value to the successor nodes in the macro data-flow graph in which the member function appears. Mentat member functions use the `rtf()` as the mechanism for returning values. The returned value is forwarded to all member functions that are data dependent on the result, and to the caller *if necessary.* In general, copies may be sent to several recipients.

Note: There must be an rtf() for every member invocation. Failure to do so can cause deadlock, or a FUTURE_STACK_OVERFLOW error.

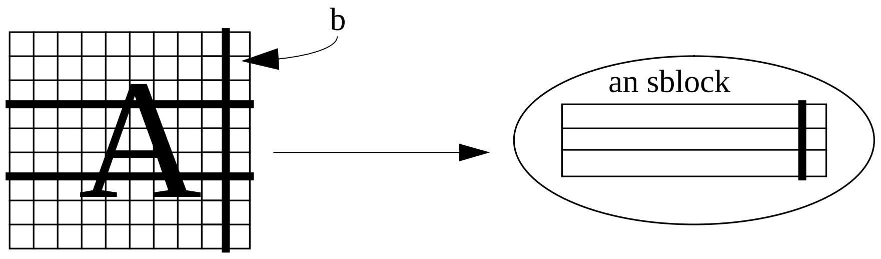
While there are many similarities between `return` and `rtf()`, `rtf()` differs from a `return` in three significant ways.

First, in **C**, before a function can `return` a value, the value must be available. This is *not* the case with an `rtf()`. Recall that when a Mentat object member function is invoked, the caller does not block, rather we ensure that the results are forwarded wherever they are needed. Thus, a member function may `rtf()` a “value” that is the result of another Mentat object member function that has not yet been completed, or perhaps even begun execution. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

Second, a **C** `return` signifies the end of the computation in a function, while an `rtf()` does not. An `rtf()` indicates only that the result is available. Since each Mentat object has its own thread of control, additional computation may be performed after the `rtf()`, e.g., to update state information or to communicate with other objects. In the message passing community this is often called send-ahead. By making the result available as soon as possible we permit data dependent computations to proceed concurrently with the local computation that follows the `rtf()`.

Third, a `return` returns data to the caller. An `rtf()` may or may not return data to the caller depending on the data dependencies of the program. If the caller does not use the

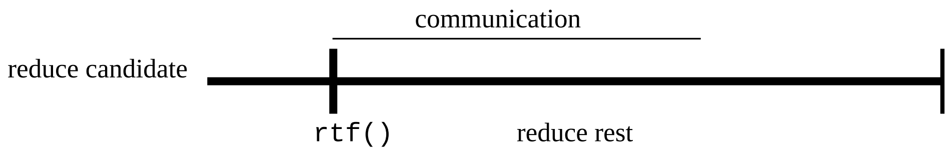
Figure 3 **Gaussian Elimination with Partial Pivoting Illustrating the Use of `rtf()`**



(a) Decomposition into `sblock`s.

```
vector*sblock::reduce(vector* pivot) {
   reduce current column using pivot
   find candidate row, it has the largest absolute
      value in current column
   reduce candidate row
   rtf(candidate row);
}
```

(b) `sblock::reduce()` pseudo-code



(c) Overlap of communication and computation with `rtf()`.

result locally, then the caller does not receive a copy. This saves on communication overhead. The next two examples illustrate these features.

Example 1. Consider a `persistent class sblock` used in Gaussian elimination with partial pivoting. In this problem, illustrated in Figure 4, we are trying to solve for x in Ax=b. The `sblocks` contain portions of the total system to be solved. The sblock member function

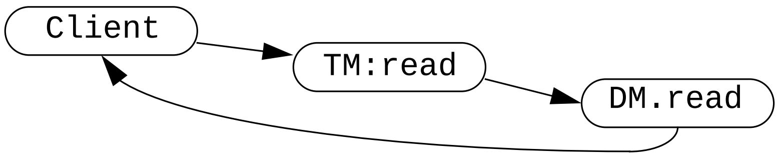
vector* sblock::reduce(vector*);

performs row reduction operations on a submatrix and returns a candidate row. Pseudo-code for the reduce operation is given in Figure 4*b*. The return value can be quickly

Figure 4 **Tail Recursion in MPL**

```
TM::read(int transaction_id, int record_number) {
   check_if_ok(transaction_id, READ, record_number);
   // Assume that check_if_ok handles errors
   rtf(DM.read(record)); // Note tail-recursive call
}
```

(a) Code fragment for Transaction Manager, `read()` member function



(b) Call graph illustrating communication for `TM::read()`

computed and returned via `rtf()`. The remaining updates to the sblock can then occur in parallel with the communication of the result (Figure 4*c*). In general, best performance is realized when the `rtf()` is used as soon as possible.

Example 2. Consider a transaction manager (TM) that receives requests for reads and writes, and checks to see if the operation is permitted. When an operation is permitted, the TM performs the operation via the data manager (DM) and returns the result. Figure 4*a* illustrates how the read operation might be implemented. In a RPC system, the record read would first be returned to the TM, and then to the user. In MPL the result is returned directly to the user, bypassing the TM (Figure 4*b*). Further, the TM may immediately begin servicing the next request instead of waiting for the result. This can be viewed as a form of distributed tail recursion, or simple continuation passing. In general, the “returned” graph may be arbitrarily complex, as in the matrix multiply example.

7.0 Select/Accept

Some form of guarded statements are provided in many modern programming languages. Examples include the select/accept statements of ADA [Ada] and guarded statements in CSP [Hoare]. Guarded statements permit the programmer to specify a set of entry points to a monitor-like construct. The guards are boolean expressions based on local variables and constants. A guard is assigned to each possible entry point. If the guard evaluates to true, its corresponding entry point is a candidate for execution. The rules vary for determining which of the candidates is chosen to execute. It is common to

Figure 5 **A Sample `mselect/maccept` Statement**

```
mselect {
      : maccept int func1(int arg1);
         break;
      : maccept int func2();
         break;
}
```

specify in the language that it is chosen at random. This can result in some entry points never being chosen.

The programmer may specify those member functions that are candidates for execution based upon a broad range of criteria. Further, the programmer may exercise scheduling control by using different priorities. The syntax for select/accept is shown below:

select_statement::	**mselect** {guard_list};
guard_list::	guard_statement; guard_list | guard_statement;
guard_statement::	[guard]:[priority] guard_action;| :[priority] guard_action;
guard_action::	**maccept** fct declarator; **break**; | **mtest** fct-declarator;statement-list; **break**;
guard ::	Boolean expression based on variables, constants, and tokens.

Note: In the current implementation, guards and priorities compile but are ignored. Tests are similarly ignored.

The select statement, and example of which is shown in Figure 5, has a similar semantics to the select statement of ADA. The availability of each guard-statement is controlled using a guard. The guards are evaluated in the order of their priority. Within a given priority level each of the guards is evaluated in some non-deterministic order. Each guard is evaluated in turn until one of the guards is true; the corresponding member function for that guard is then executed. When the function has been executed, control passes to the next statement beyond the select.

There are three types of guard-actions: accepts, tests, and non-entries. Accept is similar to the accept of ADA. Non-entries are guarded statements (**not currently supported**). They do not correspond to a member function of the class. Tests are used to test whether a particular member function has any outstanding calls that satisfy the guard. When a test guard-action is selected, no parameters are consumed. Note that there is no “else” clause as in ADA. However, using the priority options, the user can simulate one by specifying that the clause is a non-entry statement and giving the guard- statement a lower priority than all other guard-statements. Then, if none of the other guards evaluates to true, it will be chosen.

Mentat guards are more powerful than guards in traditional languages. A guard in Mentat is a boolean expression based on local variables, constants, formal parameters of the member function being guarded, and message tag information such as the sender or computation tag. Assignment statements are disallowed in guards (to prevent side effects), and accept-variables and token-variables are allowed in the expression. The use of accept variables will be expanded upon in a future release.

Priority is an integer ranging from -MAXINT to MAXINT. The default value is zero. There are two types of priority, that of the guard-statement, and that of the incoming tokens. The priority of the guard-statement determines the order of evaluation of the guards. It can be set either implicitly or explicitly. The token priority determines which call within a single guard-statement priority level will be accepted next. The token priority is the maximum of the priorities of the incoming tokens. Within a single token priority level, tokens are ordered by arrival time.

When a member function call is accepted, the current priority of the object is set to the priority of the tokens for the call. Any invoked subgraphs of the member function will have the same priority as the incoming tokens.

8.0 Parameter Passing

Mentat object member function parameter passing is call-by-value. All parameters are physically copied to the destination object. Similarly, return values are by-value. Pointers and references may be used as formal parameters and as results. However, the effect is that the **memory** object pointed to is copied. In the case of pointers the amount of data copied is determined by inspecting the class definition of the parameter (result). If the class has no `int size_of`() function defined, then `sizeof(class_name)` bytes are copied. If `size_of`() is defined, then it is invoked at run-time to determine the size of the actual parameter (result). The `size_of`() function **may not** be in-lined. While variable-size objects are supported using the above mechanism, the object must be contiguous in memory.[3] The two examples in Figure 6 illustrate the specification and use of `size_of`(). Also, refer to the $MENTAT/examples/app_misc directory for more complete examples.

9.0 Restrictions

The address space independence between Mentat objects necessitates the imposition of five restrictions on Mentat classes. These restrictions derive from the fact that instances of Mentat classes are independent objects. All communication with and between Mentat objects is via parameters; there is no shared memory.

1. The use of static member variables for Mentat classes is not allowed. Since static members are global to all instances of a class, they would require some form of

3. This restriction will be relaxed soon. The user will be permitted to specify a function, `void marshall(char*)`; that will be used to marshall arguments.

Figure 6 **Using Variable-Size Objects**

```
class string {
public:
   int size_of();
};
int string::size_of(){
   return(strlen(this)+1);
}
class dblock {
   int num_bytes;
   char data[1];
public:
   int size_of();
   char &operator[](int loc) {
      return &data[loc];
   }
   dblock (int size);
};
dblock::dblock(int size) {
   this = malloc(sizeof(int)+size);
   num_bytes=size + sizeof(int);
}
int dblock::size_of() {return num_bytes;}
persistent mentat class mfile {
// ... locals
public:
   int open(string*);
   void write(int offset;int bytes;dblock *data);
};
// Now a code fragment that uses the above definitions
int x;
dblock *data = new dblock(1024);
// Fill in data ....
mfile src;
src.create();
x=src.open((string*)”my_file”);
if (x>=0) src.write(0,1024,data);
// Etc...
```

shared memory between the instances of the object. The preprocessor detects all uses of static variables and emits an error message.

2. Mentat classes cannot have any member variables in their public definition. If data members were allowed in the public section, users of that object would need to be

able to access that data as if it were local. Any use of such variables is detected by the preprocessor. If the programmer wants the effect of public member variables, appropriate member functions can be defined.

3. Programmers cannot assume that pointers to instances of Mentat classes point to the member data for the instance. The preprocessor will not catch this.
4. Mentat classes cannot have any friend classes or functions. This restriction is necessary because of the independent address space of Mentat classes. If we permitted friend classes or functions of Mentat classes, then those friends would need to be able to directly access the private variables of instances of the Mentat class. Similarly, instances of a Mentat class cannot access each other's private data.
5. It must be possible to determine the length of all actual parameters of Mentat member functions, either at compile time or at run time. This restriction follows from the need to know how many bytes of the argument to send. Furthermore, each actual parameter of a Mentat member function must occupy a contiguous region of memory in order to facilitate the marshaling of arguments. Variable size classes must provide the member function `size_of`().

10.0 Warnings

There are a number of issues that MPL programmers must be aware of that can lead to unpredictable program behavior. First, reference and pointer arguments passed to Mentat class member functions are not preserved after the call. Consequently, the programmer must take care to first copy the arguments, if they are needed after the function invocation. Symmetrically, if a Mentat member function returns a pointer, the programmer must explicitly delete the reference when the function is finished using the value. The pointer is not deleted automatically when the function exits. If the programmer does not reclaim storage, memory leaks may result. This is a problem common to C programs. Second, virtual functions do not work on Mentat class objects that have been explicitly passed as parameter arguments. The MPL programmer is advised to consult the Mentat User's Manual to determine the implementation status of several MPL features currently not supported. Third, semantic equivalence to the sequential program is not guaranteed when persistent objects are used. This is trivially true for programs that have select/accept statements; there are no serial equivalent. Mentat guarantees only that observable data dependencies are enforced. In order to ensure semantic equivalence a sequential object should be used.

There are two cases to consider: (1) If the name of a Mentat object is passed to other Mentat objects and they access the object, then the order of access is not necessarily the same as in a sequential execution. (2) Repeated calls to the same persistent object are not necessarily executed in the same order if there is a data dependency between invocations. For example:

```
// Let A be a bound persistent Mentat object
int x, y;
x = A.op1(5);
y = A.op1(x);
```

will be executed in order. Assume `op2()` is returns a void. The sequence:

A.op2(5);
A.op2(x);

can make no guarantee on order. If the object A is defined as a sequential Mentat class, then the member functions will be executed in the expected order.

11.0 Extended C++ Examples

The standard Mentat distribution comes with a set of example Mentat classes and applications. You will find them in the “$MENTAT/examples” directory.

One example is a `regular mentat class matrix_ops` that includes functions to perform the standard matrix and vector operations including matrix multiplication and a solver that uses Gaussian elimination with partial pivoting. The example illustrates the specification of a mentat class, use of the class, and the use of a main program that uses Mentat objects. The directory includes a makefile that illustrates how to use the compiler. You may need to change the paths for the make to complete in your environment.

Another example is the traditional Fibonacci. The example is implemented with two regular Mentat classes, `fibonacci_class`, and `adder_class`. The class `adder_class` is used instead of a “+” operator because it illustrates tail recursion. The

rtf(adder.add(fib.fibonacci(n - 1), fib.fibonacci(n - 2)));

call allows the caller to exit and not wait for the result, reducing the number of objects that are instantiated at any given instant.

12.0 Summary

Mentat class definition:

new_class_def::	mentat_definition class_definition | class_definition
mentat_definition::	**persistent mentat** | **sequential mentat** | **regular mentat**
class_definition::	**class** class_name {class_interface};

Mentat member function invocation:

<Mentat_class_variable>.<member_fn> (args)

The semantics of Mentat member function invocation is call-by-value parameter passing.

Create():

The `create()` member function call creates a persistent Mentat object bound to `<Mentat_cv>.`

```
<Mentat_cv>.create()
<Mentat_cv>.create(<Mentat_cv2>)
<Mentat_cv>.create(int on_host)
```

The Mentat class variable `<Mentat_cv>` must be **persistent** or **sequential**. The `create()` member function may be overloaded by the user. If the user overloads `create(int)`, the `create(int on_host)` will be hidden.

Bind():

The `bind()` member function call binds `<Mentat_cv>` to an Mentat object instance located within scope.

```
<Mentat_cv>.bind (int scope)
<Mentat_cv> = <expression>
```

Scope is one of `SEARCH_LOCAL`, `SEARCH_SUBNET`, or `SEARCH_GLOBAL`. In the second form, binding is done implicitly where `<expression>` resolves to a Mentat object instance.

Bound():

The `bound()` member function call determines if `<Mentat_cv>` is bound to an instance, returns 0 or 1.

```
<Mentat_cv>.bound()
```

Destroy():

The `destroy()` member function call destroys persistent Mentat object bound to `<Mentat_cv>.`

```
<Mentat_cv>.destroy()
```

SELF:

The keyword `SELF` a system-defined name that the “current” Mentat object is bound to. It is available within Mentat member functions; the Mentat analogue of “this” in C++.

13.0 Programming Language Points

This is a list of rules that can help the programmer avoid common pitfalls. Most of these points stem from the use of pointers; while pointers can be used as arguments or return values, they are always handled as the full data structure.

- Data is copied *by value*.
- Data *must* be contiguous in memory.
- data must be of fixed size *or* have a *size_of()* function defined.
- Pointer arguments *do not* persist after the member function call; i.e., you cannot safely save the pointer. If you need to keep the data, copy it.
- For variable-size lists, derive a new class off of `transportable_list` as shown in $MENTAT/examples/misc/result_list.h.
- See `DD_array`, `DD_floatarray`, etc., in $MENTAT/sources/rtl/DD_array for an example of how to define variable-size one- and two-dimensional arrays of you favorite type.
- Virtual functions *do not* work as expected between Mentat object address spaces. Virtual functions rely on pointers to the function's definition; such pointers are meaningless in distributed memory systems.

14.0 References

[Stroustrup] B. Stroustrup, The C++ Programming Language, Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd Edition, 1991.

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

[Hoare] C.A.R. Hoare, "Communicating Sequential Processes," Communications of the ACM, pp. 666-677, August, 1978.