The Mentat Programming Language:
Users Manual & Tutorial

Andrew S. Grimshaw and Edmond C. Loyot

Computer Science Report No. TR-90-08
April 1990

The Mentat Programming Language: Users Manual & Tutorial

Version 1.0

Andrew S. Grimshaw
Edmond C. Loyot
Department of Computer Science
University of Virginia, Charlottesville, VA

Abstract:

An important problem facing designers of parallel systems is how to simplify the writing of concurrent applications. Mentat is an object-oriented, parallel computation system designed to meet the challenge of providing easy to use parallelism. We accomplish this goal by extending the encapsulation provided by objects to the encapsulation of parallelism. We define two forms of parallelism encapsulation supported by Mentat: *inter-object parallelism*, and *intra-object parallelism*. Users of Mentat objects are unaware of whether operations are carried out serially or in parallel. This document describes the Mentat Programming Language (MPL) Version 1.0 and its use. Version 1.0 includes a prototype MPL compiler based on the ATT 1.2 C++ compiler [23]. A later version of this technical report will describe the new MPL compiler currently under development. Completion is expected in September, 1990.

1. Introduction

One problem facing parallel systems designers is how to simplify the writing of parallel programs. Proposals range from automatic program transformation systems that extract parallelism from sequential programs [1,2], to the use of side-effect free languages [3,4], to the use of languages and systems where the programmer must explicitly manage all aspects of communication, synchronization, and parallelism [5,6]. 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 [7,8] provides a compromise solution. There are two primary aspects of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system. MPL is an object-oriented programming language based on C++ [9] 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 normally, 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 those machines more accessible to non-computer scientist researchers.

This document describes the Mentat Programming Language. Section 2 is a brief introduction to Mentat and the macro data flow model of computation. In Section 3 the MPL is described. This sections includes several examples. In Section 4 we discuss the MPL compiler. Appendix 1 is the current compiler bug list. Appendix 2 is the source listings for a set of implemented Mentat classes that perform matrix operations.

2. Mentat Overview

Mentat was designed to meet three goals. First, Mentat permits high degrees of parallelism (smaller grain than Unix processes, but larger grain than individual instructions). Second, Mentat uses a simple decentralized control mechanism in order to avoid the bottleneck of a centralized scheme and the communications overhead of some distributed schemes. Third, Mentat is easy to program, masking much of the complexity of the parallel environment from the programmer. The first two of these goals are addressed by the model of computation used by Mentat, and the third by extending the object-oriented programming paradigm to include the abstracting of parallel activity.

The model of computation underlying Mentat is the macro data-flow model [10-11]. The macro data flow model of computation is a medium-grain, data-driven model inspired by the data flow model [12-13]. A data flow program is a directed graph in which nodes are computation primitives called *actors*. *Tokens* carrying data and control information flow along the arcs from one actor to another. When tokens are present on all incoming arcs, an actor is enabled and may execute. Thus a high degree of parallelism can be achieved naturally. We were drawn to the traditional data-flow model of computation because of the intuitive ease with which it maps to distributed message passing systems and because of the high degrees of parallelism which it affords. However, the traditional data-flow model is inappropriate because of its small computation granularity [8,14] and lack of persistent data.

Macro data flow has three principle differences with traditional data flow. First, the granularity of the actors is larger and under programmer control. This provides the flexibility to choose an appropriate degree of parallelism. Second, some actors can maintain state information between executions. These actors are called *persistent actors*.

Persistent actors provide an effective way to model side effects and to reduce communication. Third, the structure of macro data-flow program graphs is not fixed at compile time. Instead, program graphs grow at run-time by elaborating actors into arbitrary subgraphs. These graph elaborations are completely local and do not affect any other actor or arc in the graph. The locality has important implications for distributed control. In Mentat, actors correspond to Mentat object member functions; each Mentat object implements a set of macro data flow actors.

The run-time system is that portion of Mentat that is responsible for the execution of Mentat programs [15]. The run-time system provides three basic services to Mentat applications. These are to instantiate Mentat objects, to construct program graphs, and to schedule program graph execution. The run-time system is not a complete operating system. Instead, the Mentat run-time system runs on top of an existing host operating system. Currently the Mentat run-time system runs on the Intel's NX/2[16] and on a network of Suns using SunOS [17].

3. The Mentat Programming Language

The Mentat Programming Language has been implemented with four goals in mind. First and foremost, the MPL is object-oriented [18-19]. We extended the usual notions of data and method encapsulation to include *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.

Second, the language constructs have a natural mapping to the macro data flow model. Furthermore, the responsibility for performing the mapping is not the programmer's. Instead, the responsibility is the compiler's and run-time system's whenever possible. In particular, the programmer does not have to make scheduling decisions, or manually construct and manage program graph execution.

Third, the concepts used as the basis of the extensions are applicable to a broad class of languages.

Fourth, the syntax and semantics of the extensions follow the pattern set by the base language, maintaining its basic structure and philosophy whenever possible.

These goals were met in the MPL by extending the C++ language in five ways. The basic idea 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. The compiler generates code to construct and execute macro data flow graphs (data dependency graphs) in which the actors 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 actors in a generated program graph may itself be transparently implemented in a similar manner by a macro data flow subgraph. Thus we obtain intra-object parallelism encapsulation; the caller only sees the member function invocation.

3.1. Extensions

There are five principle extensions to the language: *Mentat* classes, the member functions *create()* and *destroy()*, the return-to-future (*rtf()*) mechanism, the member function *main()* for each Mentat class, and guarded statements. In this section, each of the extensions is described in detail.

3.1.1. Mentat Classes

Mentat classes are the mechanism for specifying Mentat actors. Each macro data flow actor is implemented by an operation of a Mentat object, and each object in Mentat is composed of one or more actors. 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., the object operations are not computationally complex enough, should not be Mentat objects. To provide the programmer a way to control the degree of parallelism, Mentat allows both standard C++ classes and Mentat classes to be defined. 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. He may specify whether the class is persistent or regular. The syntax for Mentat class definitions is given below. (Keywords are in **boldface**.)

```
new_class_def       ::    mentat_definition class_definition |
                          class_definition
mentat_definition   ::    PERSISTENT Mentat |
                          REGULAR mentat |
                          mentat
class_definition    ::    class class_name {class_interface};
```

Persistent and regular class definitions correspond to persistent and regular objects[1]. Figure 1 illustrates a Mentat class definition.

```
PERSISTENT Mentat class class_A {
        /* Note that private variables are truly private*/
        int x,y,z;
public:
        int operation1(int arg1);
        create(int initial_x, int initial_y);
};
```

Figure 1.

Mentat classes are different from standard C++ classes in many ways. Each Mentat class is really two distinct entities: a *front-end* and a *server-end*. Instances of the front-end are called *Mentat variables*. Mentat variables are created when a user declares an instance of a Mentat class in his program, e.g. **A1** in Example 1 (Figure 6). A Mentat variable is really a name that may point to an actual instance of a Mentat object. Instances of the server-end of a Mentat class are objects that actually implement the class. We call these instances *Mentat objects*. Mentat objects are independent objects; they are address space disjoint.

Mentat variables are the mechanism through which programmers access and use Mentat objects. Mentat variables are C++ objects whose type is a Mentat front-end. Mentat variables contain a Mentat object name and incidental information and are typed according to the class of the Mentat object to which they point. Since Mentat variables are object names and not the actual representation of their corresponding Mentat object, attempts to access private object data using the address of the Mentat variable will not succeed.

Mentat variables may be either bound to a specific instance of a Mentat object or unbound. Unbound Mentat variables do not address any particular instance. When unbound names are used for regular Mentat classes the underlying system is free to choose an instance or create a new instance for each invocation of a member function. When an unbound name is used for a persistent Mentat class the system chooses an existing instance.

---

[1] Persistent objects maintain state information between method invocations, regular objects do not, i.e., regular objects are stateless.

Each Mentat object (instance of a server-end) is disjoint from all other instances in the address space. The server-ends are implemented by the code of the Mentat class definition. Mentat objects may contain other non-Mentat objects and variables. The private data of the class definition is global and persistent to each instance. Each instance has an independent thread of control, which begins at the first statement of the member function *main()*. It also has a system-wide unique name. The name is a Mentat object name and is available in the predefined Mentat variable *SELF*.

3.1.2. Mentat Class Instantiation

The issue of how to instantiate Mentat objects is an important one. Following the flavor and semantics of C++ would make it difficult to define *generic* instances. The difficulty stems in part from the fact that instances of Mentat classes are represented by their names and not by their physical selves. Only the name of the object is stored by the invoking object, not the instance itself. To illustrate this point, we consider an example in which there is a Mentat class *class_A*. There are three ways Mentat variables of *class_A* may be defined, static variables of *class_A*, auto variables of *class_A*, and heap variables of *class_A*. Furthermore, *class_A* could be a persistent or a regular class. We want to consider the problem of instantiating instances of *class_A* if we want to preserve the exact semantics of C++. In C++ static variables are instantiated at the start of the program, auto variables when the containing block of code is entered, and heap variables when they are *newed* or *malloced*. In Mentat these ways may not be appropriate for two reasons. First, regular objects may not ever need to be instantiated. For example, suppose that *class_A* is a regular Mentat class. Let *FRED* be defined to be of type *class_A*. If *FRED* is used often in expressions it may be better not to use the same instance each time. Instead we may want the underlying system to find or *create* instances at run-time, using a different instance for each invocation in order to increase parallelism. Therefore, we generally will want regular objects to be unbound to a particular instance. But if we instantiate a new instance as the usual rules call for in C++, using different instances would be incorrect.

Second, instead of creating new instances of a persistent class as called for by C++, it is sometimes more desirable to use the object as a name only, without instantiating a new version. For example, suppose there is a predefined file system object *file_system* that returns a *file_object* Mentat variable as the result of an open call. In the code fragment below we do not want the declaration of A to result in the instantiation of a new file object; we want an unbound Mentat variable created.

```
A       file_object;    // Do not want a new file created.
A = file_system.open("user-data");
```

Our solution to this problem is to specify that when a new Mentat variable is declared a new instance of the object class is not automatically instantiated. Instead, only an unbound object name of the appropriate type is instantiated.

The next issue to be considered is how the programmer creates new instances of Mentat objects. We have added two new reserved member functions for all Mentat class objects: *create()* and *destroy()*. These functions are inherited from the base class Mentat and can be overloaded by the programmer of the class. The *create()* function is used to instantiate new instances of Mentat classes. It takes as parameters user-provided initialization information. *Create()* also allows the user to specify where the new instance is to be instantiated, e.g., on a different processor, or on the same processor as some other Mentat object. The syntax is:

```
create_member     ::    create(argument_list) |
                        create(argument_list) (location_hints)
location_hints    ::    CO_LOCATE obj_name |
                        DISJOINT obj_name_list |
                        HIGH_COMPUTATION_RATIO
obj_name_list     ::    obj_name, obj_name_list |
                        obj_name
obj_name          ::    identifier
```

The programmer can specify location_hints, providing the underlying system with information that will be used in making instantiation decisions. The three values of location_hints are CO_LOCATE, DISJOINT, and HIGH_COMPUTATION_RATIO. CO_LOCATE tells the system to locate the object being created close enough to the object named by *obj_name* so that communication between the two is cheap. This usually means on the same processor. DISJOINT tells the system that the object to be created should be instantiated far away from any object in the obj_name_list because the object will usually be enabled in parallel with the objects named in the obj_name_list. It may not be possible to instantiate disjointly, if, for instance, every processor on the system has at least one object from *obj_name_list* on it. In that case the system does the best it can. HIGH_COMPUTATION_RATIO tells the system that the object to be instantiated has a particularly high computation ratio. The system can use this information to ensure that it is placed on a lightly loaded or powerful processor, even if the processor is very far away. The precise meaning of "close" and "far away" will vary depending on the algorithm used to make location decisions.

3.1.3. Return to Future (*rtf()*)

The function *rtf()* is the Mentat analog to the *return()* of C++. Its purpose is to allow Mentat member functions (actors) to return a value to the successor nodes in the macro data-flow graph in which the member function appears. The *rtf()* does not necessarily mark the end of the actor computation.

Rtf() takes two types of arguments, local variables or constants, and subgraphs. The local variables must satisfy the same restrictions that apply to Mentat arguments: they must have a determinable length, and they must be contiguous. The subgraphs are automatically generated, and their use is transparent to the programmer. Returning a subgraph using *rtf()* is the mechanism for actor subgraph elaboration. Both cases are illustrated below. The object operation *op1* occurs in the subgraph shown in Figure 2(a). The code fragment for *op1* appears in Figure 3. If *expression(input)* is TRUE, then *op1* will generate a token containing the value 5 that will be forwarded to *op1's* successor. This is illustrated in Figure 2(b). If on the other hand, *expression(input)* is FALSE, then a subgraph will be generated. The result of the new subgraph will be directed to the successor of *op1*. The subgraph of Figure 2(c) is the actual graph executed.

3.1.4. Guarded Statements

Some form of guarded statements are provided in many modern programming languages. Examples include the select/accept statements of ADA [20] and guarded statements in CSP [21]. 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 for determining which of the candidates is chosen to execute vary. It is common to specify in the language that it is chosen at random. This can result in some entry points never being chosen and is not *fair*.

A major shortcoming of this approach in languages such as ADA is that it is a non-symmetric mechanism: the caller can specify those modules with which it will interact, but the callee has no way to specify the modules from whom it will accept calls. Further, the callee has no way to specify restrictions on the parameters before rendezvous occurs.

We overcome these shortcomings in the Mentat programming language. We allow the programmer to specify those member functions that are candidates for execution based upon a broad range of criteria, and we provide the

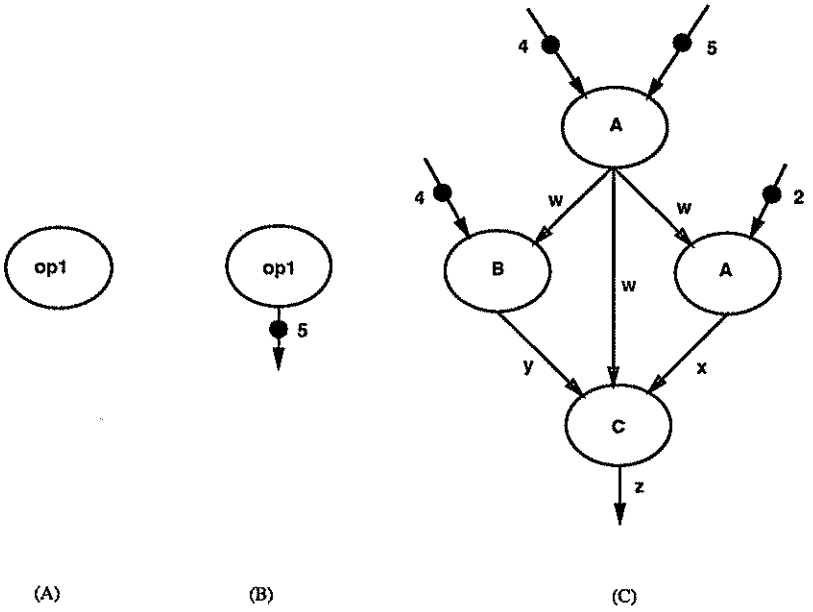

Figure 2. Initial subgraph.

```
if expression(input) {
        rtf(5);
}
else {
        w=A.operation1(4,5);
        x=A.operation1(w,2);
        y=B.operation1(4,w);
        z=C.operation1(y,w,x);
        rtf(z);
};
```

Figure 3. Code for *op1*.

programmer of objects with scheduling control to give different priorities to different candidates for execution. The syntax for select/accept is shown below.

select_statement	::	**select** { guard_list };
guard_list	::	guard_statement;guard_list |
		guard_statement;
guard_statement	::	guard:guard_action;|
		guard:[priority] guard_action;|
		guard_action;|
		[priority] guard_action;
guard_action	::	statement-list;break;|
		accept fct-declarator; statement-list;break; |
		test fct-declarator;statement-list;break;
guard	::	Boolean expression based on variables, constants, and tokens.
		Exact syntax discussed later.

The select statement has a similar semantics to the select statement of ADA. It consists of a list of guard-statements, corresponding to the ADA accept statement. 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 statement-list for that guard is then executed. When the statement-list has been executed, control passes to the next statement beyond the select. Note that the fct-declarator portion of the guard_action determines the operation. The fct-declarator is not actually executed. It only provides information for the compiler.

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. 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 much more powerful than guards in traditional languages such as ADA. 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. The ability to selectively receive messages based upon their contents is also available in PLITS [22]. The guard syntax is:

guard	::	(NOMATCH && guard1) | guard1
guard1	::	(guard1) |
		unary-operation guard1 |
		guard1 binary-operation guard1 |
		value |
value	::	constant | variable | accept-variable |
		token-variable | expression
token-variable	::	arg-ident.arg-field
arg-ident	::	**arg1** | **arg2** | ... | **argN** | accept-variable
arg-field	::	c-tag | from | priority | length

Guards are similar to *expressions* in C++, except that assignment statements are disallowed in guards (to prevent side effects), and accept-variables and token-variables are allowed in the expression.

Accept-variables are defined by the formal arguments of the member function of the *accept* or *test* which this expression guards. As such, an accept-variable is an identifier whose scope is the entire guard-statement in which it

occurs. For example the accept-variables *account* and *amount* are active in the entire fragment in Figure 4.

```
(amount < 200) : accept withdrawal(int account, int amount);
                 if account_exists(account)
                        withdrawal(account, amount);
                 else
                        error(NO_SUCH_ACCOUNT, account);
                 break;
```

Figure 4.

Token-variables are fields of the arriving messages that are not part of the user data of the message, i.e., they are extra control information used by the underlying system. The token-variables are accessible in a read-only fashion by the applications programmer. There are four token-variables for each message, *source*, *c-tag*, *priority*, and *length*. The *source* field is the name of the Mentat object from which the token has come. The *c-tag* is the computation tag of the token. *Priority* is the priority of the token, and *length* is the length of the data part of the token. These fields are provided to the user so that finer control may be obtained. However, it is not expected that they will be frequently used.

There is one set of token-variables for each accept-variable in the guard-statement. The token-variables have the same scope as accept-variables. They can be named by either their corresponding accept-variable name with a field name suffix, or by the argument number and a suffix, e.g., arg1.c-tag.

By default, tokens must have matching c-tag fields to be candidates for matching. This is accomplished by having an implicit

(arg1.c-tag == arg2.c-tag == ... == argN.c-tag)

ANDed to each guard. However, there are circumstances under which it might be desirable to circumvent this constraint. To do so the programmer adds the keyword *NOMATCH* as the first clause in the guard, e.g.,

```
struct srec1 {
        int     account;
        ....    // Some other stuff, application specific
        };
struct srec2 {
        ....    // Some stuff, application specific
        int     account;
        };
        ...
select {
        (NOMATCH && (rec1.account == rec2.account))
                :accept debit(s_rec_1 rec1,s_rec_2 rec2)
        ...
};
```

Figure 5.

Figure 5 illustrates the capability to match on fields other than c-tag. Each possible matching pair of tokens is examined to determine if it satisfies the guard. Note that it will likely involve much more overhead to check a guard when c-tag matching is turned off.

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.

3.1.5. The Member Function *Main()*

The member function *main()* is a reserved function name for Mentat classes. It is the initial thread of control for new instances of a Mentat class. There are two methods of constructing *main* functions. First, the programmer may specify a member function *main()* for each persistent class in the class definition. The syntax is:

class-name::*main()* { *function body* }

The function body may be any sequence of extended C++ statements. Second, if the programmer elects not to create a member function *main()* then the preprocessor will construct one for the class. It is of a very simple form, based on either a persistent class template or a regular class template.

The persistent class template has two stages. In the first stage a select/accept is used to accept the create call for the object, if such a create exists. Then, once the create function has been executed, a loop is entered in which the sole statement is a select/accept. Within this select/accept statement there is an accept for each member function in the class interface. In this manner each member function of the class is made equally available to external objects.

The regular class template is much simpler. It consists of a call to the local constructor followed by a single select/accept statement. Like the persistent class template, the select/accept statement contains an accept for each member function in the class interface.

The member function *main()* is the initial entry point for all instances of Mentat objects. To instantiate a new instance of a Mentat class at run-time, the underlying system first allocates storage for the object and then starts the object running by starting the function *main()*. It may leave *main()* and enter another section of the object. The thread of control is not required to ever return to *main()*.

3.1.6. 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 tokens; there is no shared memory. First, 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 shared memory between the instances of the object. The preprocessor detects all uses of static variables and emits an error message. Second, 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. Third, programmers cannot assume that pointers to instances of Mentat classes point to the member data for the instance. The preprocessor will not catch this. Fourth, Mentat classes cannot have any friend classes or functions. This restriction is neces-

sary 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. Fifth, 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()*.

3.2. Extended C++ Examples

This section contains three examples that illustrate the use of the Mentat language extensions. Comments on the semantics of the Mentat statements have been included as comments on the code.

The first example (Figure 6) illustrates the definition and use of both persistent and regular Mentat classes. Three Mentat classes are defined (**A**, **m_B**, **C**), and one non-Mentat class (**B**). Note that the Mentat class **m_B** is derived from the non-Mentat class **B**.

```
PERSISTENT Mentat class A {
        /* Note that private variables are truly private*/
        int x,y,z;
public:
        int operation1(int arg1);
        create(int initial_x, int initial_y);
};

class B {
        int i,j,k;
public:
        int operation1(int arg1);
        B(int initial_x, int initial_y);
};

PERSISTENT Mentat class m_B : B {
        // Note that this is a derived Mentat class, it will inherit all of the
        // functions and interface of B. However, if this inheritance causes Mentat_B to
        // violate the Mentat class restrictions, a compiler error will result.
public:
        constructor-to-create  // This macro constructs create from the base class constructors.
};

Mentat class C {
        int x,y,z;
public:
        int operation1(int arg1);
};

{       int result1,result2;
        A A1;           // Instantiates a new Mentat variable of class A
                        // No new Mentat object is instantiated
        result1 = A1.operation1(5);
                        // An existing instance of Mentat class A is chosen
                        // by the system to service the call.
        B B1(1,2);      // Instantiates a contained object of class B
        m_B B2;                 // Instantiates a new Mentat variable of class m_B
        C C1;           // Instantiates a new Mentat variable of class C
        A1.create(3,4);         // Instantiates a new Mentat object of class A
        B2.create(1,2);         // Instantiates a new Mentat variable of class m_B
        result1 = A1.operation1(5);
                        // The Mentat object instantiated above is used to service the call.
        result2 = C1.operation 1(5);
                        // The system either creates a new instance to
                        // service the call or uses an existing instance.
};
```

Figure 6

The second example (Figure 7) illustrates the use of select/accept statements that contain both local variables and message values in guards.

```
PERSISTENT Mentat class account {
        int balance=0;
public:
        BOOLEAN withdrawal(int w_amount);
        BOOLEAN deposit(int amount);
};
void account::main() {
        while (TRUE) {
                select {
                        ((balance - w_amount) > 0) : accept withdrawal(w_amount);
                                withdrawal(w_amount);
                                break;
                        (amount > 0) : accept deposit(amount);
                                deposit(amount);
                                break;
                };
        };
};
```

Figure 7

The third example (Figure 8) shows a solution to the Readers & Writers problem with priority given to the writers. It illustrates the main() function, select, guards, and priority. The functions writer_get(), writer_return(), reader_get(), and reader_return are not shown.

```
PERSISTENT Mentat class RW {
        int writer, num_readers, done;
public:
        void create();
        void destroy();
        int writer_get();
        int reader_get();
        int reader_return();
        int writer_return();
};
void RW::create();
{       writer=0;
        num_readers=0;
        done=0;
        rtf();
};
void RW::main()
{       select {
                :accept         create();
                                create();
                                break;
        };
        while (!done) {
                select {
                        (!writer):[1] accept writer_get();
                                while (num_readers > 0) {
                                select {
                                accept reader_return();
                                reader_return();
                                break;
                                }; };
                                writer_get();
                                break;
                        (!writer):[0] accept reader_get();
                                reader_get();
                                break;
                                :[1] accept writer_return();
                                writer_return();
                                break;
                                :accept reader_return();
                                reader_return();
                                break;
                                :accept destroy();
                                done = 1;
                                rtf();
                                break;
                }; }; };// end select, end while loop, end main()
```

Figure 8

4. Using The MPL Compiler

In this section we briefly describe the current version of the MPL compiler. This version is based upon the ATT 1.2 C++ compiler. The MPL compiler consists of two passes, *sfront*, and *cfront*. Sfront scans input files and generates C++ code that is passed to cfront. Sfront reads Mentat class definitions and generates server-end code. In particular sfront is responsible for generating *main()* functions based upon the persistent and regular class templates. Cfront is responsible for generating the code to do data flow detection. Cfront generates *C* code, as does the ATT compiler. The version described is an early version. The current version is not as sophisticated as we would like, and does not support the select/accept construct. We are working on a new compiler that is not based on the ATT code.

4.1. Invoking the MPL Compiler

The MPL Compiler is invoked by executing the program MC (Mentat Compiler). For example, the MPL program matrix.c can be compiled by the command:

```
MC matrix.c
```

MC accepts the same flags as the standard C++ compiler CC (see the man page for CC for details).

4.2. Structuring MPL Programs

There are two components to a Mentat class definition. The first component is a header file that must be included by all programs that use the Mentat class. The other component is a file that will be compiled into the server for the class. This is the implementation file.

The header file is just the C++ class definition with three additions. First, in all Mentat class definitions the keyword mentat must immediately proceed the C++ keyword class. This indicates to the compiler that this is a Mentat class definition, not a standard C++ class definition. Second, all Mentat classes must inherit the public member functions from the class mentat_object. This allows the compiler to correctly generate Mentat member function calls and also to handle create(), bind() and destroy() correctly. Third, all Mentat classes must include an inline constructor of the following form:

class-name() {set_object_name(&i_name,"*class-name*");}

where *class-name* is the actual name of the Mentat class. The constructor must be the last member function defined. This constructor identifies the Mentat class to the rest of the system.

The implementation file is the same as a C++ implementation of the class with two differences. Once again the keyword mentat must appear immediately before the C++ keyword class. Also, the *return()* statement is replaced by the *rtf()* statement in the member function definitions (for an explanation of the meaning and use of *rtf()* see section 3.1.3). As an example consider the following:

```
Mentat class fib_class : public mentat_object
{
public:
int     fibonacci(int n);
        fib_class() {set_object_name(&i_name,"fib_class");}
};
```

Figure 9. Header File.

```
#include "reg_class.h"
#include "fib_class.h"
Mentat class fib_class {
public:
        int     fibonacci(int n);
};

int fib_class::fibonacci(int n) {
        reg_class       adder;
        fib_class       A, B;
        int             result,
                        n1,
                        n2;
        if(n==0 || n==1) {
                result = n;
                new_rtf(result);
        }
        else {
                n1 = n-1;
                n2 = n-2;
                new_rtf(adder.operation1(A.fibonacci(n1),B.fibonacci(n2)));
        }
}
```

Figure 10. Implementation File.

For the meaning and use of persistent see section 3.1.1. In the current implementation there can only be one Mentat class definition per file. The need for a different class declarations in the header and implementation files will be eliminated in a future version of the compiler.

4.3. MC Intermediate Files

MC has two main phases. In the first phase MC reads the source file and builds the MPL code for a Mentat server for the specified class. In the second phase MC translates MPL code for the above server.

The result of the first phase is a file containing the server code. The named of this file is obtained by prepending _ms_ (for mentat server) to the source file name. The current implementation leaves these _ms_ files around for debugging purposes.

The C code for the translation performed in phase two can also be useful for debugging. This code can be obtained by invoking the MC compiler with the -Fc option (see the C++ man page for more details of the -Fc option) and redirecting the output to a file. For example, the following command will compile the MPL program matrix.c and leave a file called _ms_matrix.c in the current working directory:

```
MC matrix.c
```

The file _ms_matrix.c will contain the MPL code for the server matrix.c. The command:

```
MC -Fc matrix.c > matrix.translation
```

Will generate the _ms_matrix.c file as above but will also produce the file matrix.translation. This file contains the C code translation of _ms_matrix.c. The -Fc option does not create an executable or object file.

Appendix 1 : Current Implementation Problems

The current implementation has several idiosyncrasies, these will be resolved in future versions.

- All member functions of Mentat classes must have at least one argument, except for the constructor discussed in section 4.2. This means that if the user's code contains member functions without arguments she must add a dummy argument. For example, the member function declaration:

```
        void op1();
would become:
        void op1(int dummy_argument);.
```

- MC will not currently detect member variables as arguments to Mentat object invocations or as arguments to *rtf*. To get around this problem assign the member variable to a local then use the local in the invocation.

- MC sfront one problems. In the current implementation phase one does not actually parse the source code. It is hardcoded to recognize the Mentat class definition constructs. Thus, stray or extra semi-colons can cause problems. This situation will be remedied soon by rewriting phase one to actually parse the input.

- The select/accept and guarded statement constructs are not currently supported.

- The member function *main()* is not currently supported.

- An instance ,*x*, of a user defined class that occurs on the *LHS* of a mentat expression is called a PRV (potential result variable). A problem occurs when a member (function or variable) of *x* is used in an expression that is not on the *RHS* of an assignment. This situation will lead to incorrect data. For example:

```
x=A.op1(...); // x is a PRV.
x.foo();      // Will yield incorrect data.

x=A.op1(...);
t=x.foo();    // Works correctly.

x=A.op1(...);
z=x;
x.foo();      // Also works correctly.
```

- MC will not detect PRV's inside the expression part of *if*, *while*, *repeat* and *for* statements. The expression part of these statements is where the test is done. For example:

```
if (expression part) ...
while (expression part) ...
for (expression part) ...
```

Appendix 2 : Matrix Class Implementation

The following code is an example of real, working, Mentat code. There are two classes defined, the class `work_class`, and `matrix_class`. They implement a collection of matrix operations. The member functions of `matrix_class` partition the operational work up into calls to instances of `work_class`. This code compiles and runs and was used as a member of a set of benchmark and validation programs. Performance figures are available in [15]. A description of the complete set of system validation/performance programs programs can be found in [24]. The `work_class` code is given first, followed by the `matrix_class` code.

```
                    /*work_class.c*/
#include "app_misc.h"
#include "work_class.h"

Mentat class work_class {
public:
     loc_matrix*    scal_add_work(loc_matrix* mat, float scalar);
     loc_matrix*    scal_mult_work(loc_matrix* mat, float scalar);
     loc_matrix*    add_work(loc_matrix* mat1, loc_matrix* mat2);
     loc_matrix*    mult_work(loc_matrix* mat1, loc_matrix* mat2);
};

loc_matrix* work_class::scal_add_work(loc_matrix* mat, float scalar) {

     int       x, j;
     float          *rrptr, *matptr;
     loc_matrix     *ret_res;

     ret_res = new loc_matrix(mat->num_col(), mat->num_row());
     for (x=0;x<mat->num_row();x++) {
          rrptr = ret_res->get_r_ptr(x);
          matptr = mat->get_r_ptr(x);
          for (j=0;j<mat->num_col();j++) {
               *(rrptr + j) = *(matptr + j) + scalar;
          }
     }
     new_rtf(ret_res);
     delete ret_res;
}

loc_matrix* work_class::scal_mult_work(loc_matrix* mat, float scalar) {
     int       x, j;
     float          *rrptr, *matptr;
     loc_matrix     *ret_res;

     ret_res = new loc_matrix(mat->num_col(), mat->num_row());
     for (x=0;x<mat->num_row();x++) {
          rrptr = ret_res->get_r_ptr(x);
          matptr = mat->get_r_ptr(x);
          for (j=0;j<mat->num_col();j++) {
               *(rrptr + j) = *(matptr + j) * scalar;
          }
     }
     new_rtf(ret_res);
     delete ret_res;
}
```

```
loc_matrix* work_class::add_work(loc_matrix* mat1, loc_matrix* mat2) {

     int       x, j;
     float          *rrptr, *mat1ptr, *mat2ptr;
     loc_matrix     *ret_res;

     if(mat1->num_col() != mat2->num_col() ||
        mat1->num_row() != mat2->num_row()) {
          printf("ERROR: Can only add matricies of the same order.0);
          exit(1);
     }

     ret_res = new loc_matrix(mat1->num_col(), mat1->num_row());

     for (x=0;x<mat1->num_row();x++) {
          rrptr = ret_res->get_r_ptr(x);
          mat1ptr = mat1->get_r_ptr(x);
          mat2ptr = mat2->get_r_ptr(x);
          for (j=0;j<mat1->num_col();j++) {
               *(rrptr + j) = *(mat1ptr + j) + *(mat2ptr + j);
          }
     }
     new_rtf(ret_res);
     delete ret_res;
}

loc_matrix* work_class::mult_work(loc_matrix* mat1, loc_matrix* mat2) {
     int       x, i, j, dim;
     float          *fp, *rrptr, *mat1ptr;
     loc_matrix     *ret_res;

     if(mat1->num_col() != mat2->num_row()) {
          printf("ERROR: Can only multiply matricies when the number0);
          printf("of columns in [A] equal the number of rows in [B]0);
          exit(1);
     }
     fp = mat2->get_r_ptr(0);
     dim = mat2->num_row();
     ret_res = new loc_matrix(mat1->num_col(), mat1->num_row());

     for (x=0;x<mat1->num_row();x++) {
          rrptr = ret_res->get_r_ptr(x);
          mat1ptr = mat1->get_r_ptr(x);
          for (j=0;j<dim;j++) {
               for (i=0;i<dim;i++) {
                    *(rrptr + j) += *(mat1ptr + i) * fp[i*dim+j];
               }
          }
     }
     new_rtf(ret_res);
     delete ret_res;
}
```

```
     /*   matrix_class.c*/

#include "app_misc.h"
#include "work_class.h"
#include "matrix_class.h"

#define MAX_PIECES 33

Mentat class matrix_class {
public:
     loc_matrix*    scal_add_mat(loc_matrix* mat, float scalar, int pieces);
     loc_matrix*    scal_mult_mat(loc_matrix* mat, float scalar, int pieces);
     loc_matrix*    add_mat(loc_matrix* mat1, loc_matrix* mat2, int pieces);
     loc_matrix*    mult_mat(loc_matrix* mat1, loc_matrix* mat2, int pieces);
};

loc_matrix* matrix_class::scal_add_mat(loc_matrix* mat, float scalar, int pieces) {

     int            dim, i, rows=0, no_rows=0, prev_rows=0;
     work_class     worker[MAX_PIECES];
     loc_matrix     *ret_res[MAX_PIECES],
                    *pm[MAX_PIECES],
                    *answer, *temp;
     float          *s_p, *d_p;

     for (i=0;i<pieces;i++) {
          worker[i].create();
     }

     dim = mat->num_row();

     for (i=0;i<pieces;i++) {
          rows=dim/pieces;
          if (dim%pieces>i) rows++;
          pm[i] = new loc_matrix(mat->get_r_ptr(no_rows), dim, rows);
          ret_res[i] = worker[i].scal_add_work(pm[i], scalar);
          delete pm[i];
          no_rows += rows;
     }

     answer = mat;
     for (i=0;i<pieces;i++) {
          temp = ret_res[i];
          s_p = temp->get_r_ptr(0);
          d_p = answer->get_r_ptr(prev_rows);
          no_rows = sizeof(float)*dim*temp->num_row();
          bcopy((char*)s_p, (char*)d_p, no_rows);
          prev_rows += temp->num_row();
          delete temp;
     }
     new_rtf(answer);
}
```

```
loc_matrix* matrix_class::scal_mult_mat(loc_matrix* mat, float scalar, int pieces) {

     int            dim, i, rows=0, no_rows=0, prev_rows=0;
     work_class     worker[MAX_PIECES];
     loc_matrix     *ret_res[MAX_PIECES],
                    *pm[MAX_PIECES],
                    *answer, *temp;
     float          *s_p, *d_p;

     for (i=0;i<pieces;i++) {
          worker[i].create();
     }

     dim = mat->num_row();

     for (i=0;i<pieces;i++) {
          rows=dim/pieces;
          if (dim%pieces>i) rows++;
          pm[i] = new loc_matrix(mat->get_r_ptr(no_rows), dim, rows);
          ret_res[i] = worker[i].scal_mult_work(pm[i], scalar);
          delete pm[i];
          no_rows += rows;
     }

     answer = mat;
     for (i=0;i<pieces;i++) {
          temp = ret_res[i];
          s_p = temp->get_r_ptr(0);
          d_p = answer->get_r_ptr(prev_rows);
          no_rows = sizeof(float)*dim*temp->num_row();
          bcopy((char*)s_p, (char*)d_p, no_rows);
          prev_rows += temp->num_row();
          delete temp;
     }
     new_rtf(answer);
}
```

```
loc_matrix* matrix_class::add_mat(loc_matrix* mat1, loc_matrix* mat2, int pieces){

     int            dim, i, rows=0, no_rows=0, prev_rows=0;
     work_class     worker[MAX_PIECES];
     loc_matrix     *ret_res[MAX_PIECES],
                    *pm[MAX_PIECES],
                    *wm[MAX_PIECES],
                    *answer, *temp;
     float          *s_p, *d_p;

     for (i=0;i<pieces;i++) {
          worker[i].create();
     }

     dim = mat1->num_row();

     for (i=0;i<pieces;i++) {
          rows=dim/pieces;
          if (dim%pieces>i) rows++;
          pm[i] = new loc_matrix(mat1->get_r_ptr(no_rows), dim, rows);
          wm[i] = new loc_matrix(mat2->get_r_ptr(no_rows), dim, rows);
          ret_res[i] = worker[i].add_work(pm[i], wm[i]);
          delete pm[i];
          delete wm[i];
          no_rows += rows;
     }
     answer = mat1;
     for (i=0;i<pieces;i++) {
          temp = ret_res[i];
          s_p = temp->get_r_ptr(0);
          d_p = answer->get_r_ptr(prev_rows);
          no_rows = sizeof(float)*dim*temp->num_row();
          bcopy((char*)s_p, (char*)d_p, no_rows);
          prev_rows += temp->num_row();
          delete temp;
     }
     new_rtf(answer);
}
```

```
loc_matrix* matrix_class::mult_mat(loc_matrix* mat1, loc_matrix* mat2, int pieces){

     int            dim, i, rows=0, no_rows=0, prev_rows=0;
     work_class     worker[MAX_PIECES];
     loc_matrix     *ret_res[MAX_PIECES],
                    *pm[MAX_PIECES],
                    *answer, *temp;
     float          *s_p, *d_p;

     for (i=0;i<pieces;i++) {
          worker[i].create();
     }

     dim = mat1->num_row();

     for (i=0;i<pieces;i++) {
          rows=dim/pieces;
          if (dim%pieces>i) rows++;
          pm[i] = new loc_matrix(mat1->get_r_ptr(no_rows), dim, rows);
          ret_res[i] = worker[i].mult_work(pm[i], mat2);
          delete pm[i];
          no_rows += rows;
     }

     if(mat1->num_col() != mat2->num_col() ||/*not square*/
           mat1->num_row() != mat2->num_row())
          answer = new loc_matrix(dim, dim);
     else
          answer = mat1;

     for (i=0;i<pieces;i++) {
          temp = ret_res[i];
          s_p = temp->get_r_ptr(0);
          d_p = answer->get_r_ptr(prev_rows);
          no_rows = sizeof(float)*dim*temp->num_row();
          bcopy((char*)s_p, (char*)d_p, no_rows);
          prev_rows += temp->num_row();
          delete temp;
     }
     new_rtf(answer);

     if(mat1->num_col() != mat2->num_col() ||/*not square*/
           mat1->num_row() != mat2->num_row())
          delete answer;
}
```

References

[1] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, "Cedar-A Large Scale Multiprocessor,"*Proceedings of the 1983 International Conference on Parallel Processing*, pp. 524-529, IEEE, 1983.

[2] D. Kuck, R. Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *ACM Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp. 207-218, January, 1981.

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

[4] J. R. McGraw,"The VAL Language: Description and Analysis," *ACM Transactions on Programming Languages and Systems*, pp. 44-82, vol. 4, no. 1, January, 1982.

[5] G. R. Andrews, and F. B. Schneider, "Concepts and Notions for Concurrent Programming," *ACM Computing Surveys*, pp. 3-44, vol. 15, no. 1, March, 1983.

[6] R. E. Filman, and D. P. Friedman, *COORDINATED COMPUTING Tools and Techniques for Distributed Software*, McGraw-Hill Book Company, New York, 1984.

[7] A. S. Grimshaw, and J. W. S. Liu, "Mentat: An Object-Oriented Data-Flow System," *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference*, ACM, pp. 35-47, October, 1987.

[8] A. S. Grimshaw, "Mentat: An Object-Oriented Macro Data Flow System," University of Illinois, TR UIUCDCS-R-88-1440, June, 1988.

[9] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

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

[11] J.W.S. Liu and A. S. Grimshaw, "A Distributed System Architecture Based on Macro Data Flow Model," *Proceedings Workshop on Future Directions in Architecture and Software*, South Carolina, May 7-9, 1986.

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

[13] A. H. Veen, "Dataflow Machine Architecture," *ACM Computing Surveys*, pp. 365-396, vol. 18, no. 4, December, 1986.

[14] J. L. Gaudiot, 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.

[15] A. S. Grimshaw,"The Mentat Run-Time System: Support for Medium Grain Parallel Computation," *Proceedings of the Fifth Distributed Memory Computing Conference*, Charleston, SC., April 9-12, 1990, also available as, Department of Computer Science Technical Report TR 90-09, University of Virginia.

[16] Intel Corporation, "iPSC/2 USER'S GUIDE," Intel Scientific Computers, Beaverton, OR, March 1988.

[17] Sun Microsystems Inc., SunOS Users Manual, Mountain View, CA, 1988.

[18] P. Wegner,"Dimensions of Object-Based Language Design," *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference*, ACM, pp. 168-182, October, 1987.

[19] B. Stroustrup,"What is Object-Oriented Programming?," *IEEE Software*, pp. 10-20, May, 1988.

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

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

[22] J. A. Feldman,"High Level Programming for Distributed Computing," *Communications of the ACM*, pp. 353-368, vol. 22, no. 6, January, 1979.

[23] AT&T,"Unix (tm) System V AT&T C++ Translator Release Notes"

[24] G. Cheek,and A. S. Grimshaw,"System Validation and Performance Analysis of the Mentat Run-Time System," Technical Report in progress, Department of Computer Science, University of Virginia, Charlottesville, VA.