Exploiting Sequential Debuggers in a Parallel Environment: An Introduction to the Mentat Assistant Debugger

Anh Nguyen-Tuong

Andrew S. Grimshaw

Exploiting Sequential Debuggers in a Parallel Environment: An Introduction to the Mentat Assistant Debugger

Anh Nguyen-Tuong

Andrew S. Grimshaw

1.0 Introduction

The traditional sequential technique of cyclic debugging — setting breakpoints to stop a program, examining the program state, recompiling the program and re-executing it until the errors are corrected — does not work well in a parallel environment if programmers simply attach a debugger to concurrently running processes. For programs with hundreds or thousands of concurrent components, this method would be unmanageable and would quickly turn into an exercise in frustration and futility. Clearly, a better way is needed to assist programmers in debugging their parallel code. Unfortunately, the alternative is oftentimes to rely on the judicious insertion of print statements to pinpoint errors.

Our philosophy in designing debugging tools for Mentat [2][3][4] — a high performance, object-oriented parallel processing system — is that debugging should be decoupled from the parallel environment. We believe that programmers are more productive if they can use the debugging environment with which they are already familiar. After all, why should users learn a new debugging tool when they have invested so much time in learning a sequential debugger?

The Mentat Assistant Debugger (MAD) is a set of tools that enables programmers to debug their Mentat applications with the debugger of their choice. MAD supports a style of debugging known as post-mortem debugging: debugging occurs after a Mentat program runs to completion or until an error occurs. MAD is based on the record and replay [5]

technique and consists of two phases. In the recording phase, objects that comprise a Mentat application log their incoming messages to a file. In the playback phase, objects can faithfully reproduce their original execution by extracting the appropriate messages from the log file. The programmer can then replay a specific object, i.e. reproduce its execution, under the control of a sequential debugger and use the traditional cyclic debugging technique.

Though we will focus on debugging in this technical report, MAD can also be used as a generic playback tool to improve and modify the implementation of Mentat objects as well as perform optimizations. The power of MAD lies in its playback capabilities and the fact that programmers can focus on a single object and execute it in complete isolation from other objects.

MAD is simple to use and requires little intervention. The programmer does not invoke special functions to record or playback messages as these capabilities are transparently embedded into all Mentat objects.

MAD does not handle all classes of bugs. It is best suited for Bohrbugs [1]: repeatable errors which usually manifest themselves by a memory fault, a floating point exception, or incorrect results. MAD is not designed to catch timing dependent errors, or Heisenbugs [1].

This technical report describes the interface and implementation of the Mentat Assistant Debugger. We assume that the reader is familiar with Mentat and provide only a brief introduction to Mentat in Section 2.0. In Section 3.0, we show the interface to MAD. In Section 4.0 we describe the implementation of the recording and playback phases and conclude in Section 5.0.

2.0 Mentat System

Mentat is an object-oriented parallel processing system designed to simplify the task of writing portable, high-performance, parallel application software. The fundamental objectives of Mentat are to (1) provide easy-to-use parallelism, (2) facilitate the portability of applications across a wide range of platforms, and (3) achieve high performance. The first two objectives are addressed through Mentat’s underlying object-oriented approach: high-level abstractions mask the

complex aspects of parallel programming, i.e. communication, synchronization, and scheduling.

Programmers write Mentat applications in the Mentat Programming Language (MPL [6]) — an extended C++ designed to simplify the task of programming parallel applications. 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 and possess a name, a thread of control, an address space and a unique identifier (Mentat UID). Mentat objects are address-space disjoint and typically implemented as heavyweight Unix processes. Programmers use Mentat objects much as they would any other C++ objects.

The MPL compiler transforms the user's MPL source code and inserts statements to interact with the Mentat run-time libraries in order to automatically handle synchronization, communication and scheduling on behalf of the user. The implication for debugging is that users must work with the modified code and not their original source code. This situation is similar to debugging C++ programs where the C++ pre-processor transforms the code into C[1].

3.0 The Mentat Assistant Debugger

Prior to the advent of MAD, debugging in Mentat mainly consisted of inserting print statements in the objects that contained errors in order to pinpoint the bug, making changes to the source code, recompiling the objects, and re-executing the entire application to determine whether the errors were corrected. The problem with this method is that it is not possible to trace through the execution of individual objects and examine their private state. In addition, print statements from Mentat objects were routed to the same console and the resulting race conditions often resulted in a meaningless jumble of sentence fragments[2]. This whole process was unnecessarily time consuming as users had to modify and recompile their objects just to find bugs. Moreover, users had to run the entire application even if the bugs were localized to a single object.

1. There now exist debuggers such as CC_ObjectCenter that work directly with C++.
2. This is a common situation in parallel processing systems.

Debugging applications with MAD consists of two phases. In the recording phase, Mentat objects record their incoming messages to a log file before processing them normally. At the end of the recording phase, the log file produced thus contains the history of all inter-object communication.

During playback, an individual object can reproduce its execution by replaying its original message history. Our assumption is that the behavior of an object is solely determined by its incoming messages and the receipt order of these messages. Thus, an object now "receives" messages from the log file instead of receiving them from other objects. In addition, outgoing messages are intercepted and never sent as they do not directly affect the behavior of the sending object[3].

To debug an object, programmers simply run the object in playback mode under the control of a sequential debugger. Print statements are no longer necessary as the sequential debugger can be used to pinpoint the bug with such traditional techniques as setting breakpoints, watching variables, etc. While this technical report focuses on debugging, the playback capability can also be used for performance optimization.

The rest of this section describes the interface to the recording (Section 3.1) and playback phases (Section 3.2) as well as the compiler flags needed to use debuggers efficiently (Section 3.3).

3.1 Recording phase

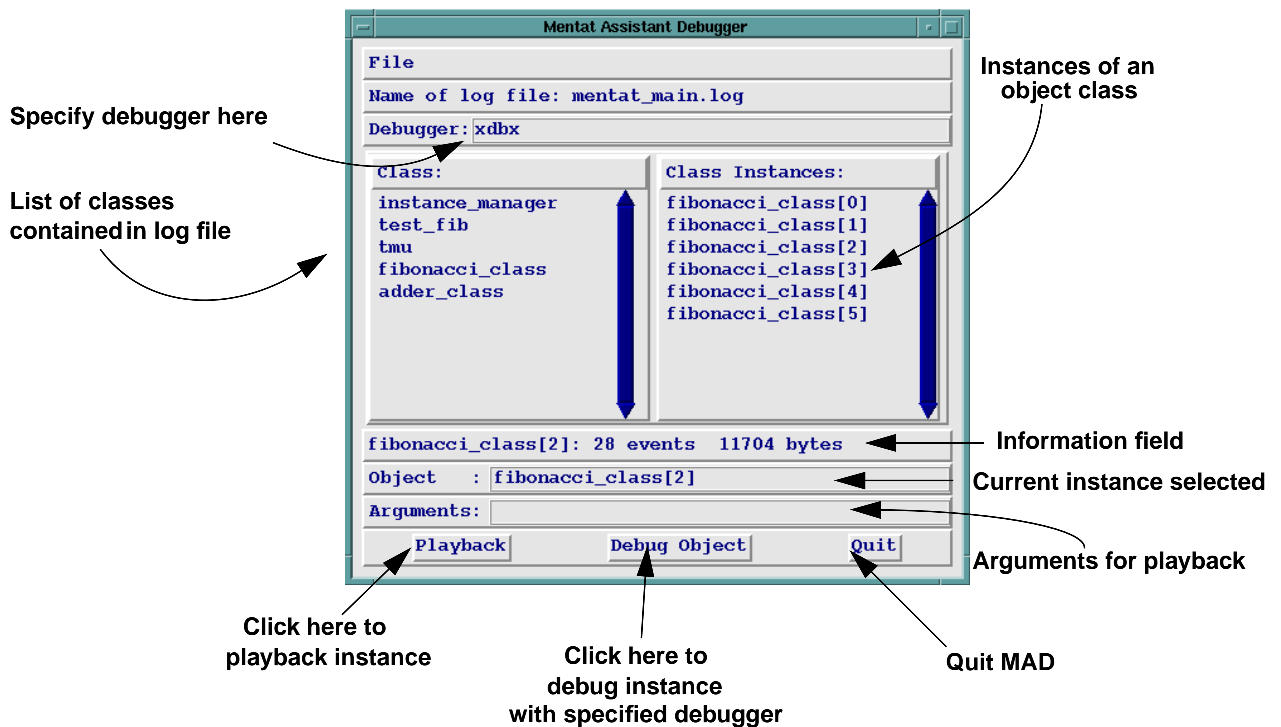
The recording phase is simple to use. Any Mentat application can be recorded by prefixing its invocation with the command `mad_record`. For example, if the original Mentat program was invoked with: `test_fib 10`, then the modified command would be: `mad_record test_fib 10`. All Mentat objects created by `test_fib` or its children execute in record mode. The output of the recording phase is a single log file containing messages received by all Mentat objects for a given application.

3. This is not strictly true. An object can send a message to itself thereby affecting its own behavior. However, this message would be recorded and played back like any other messages.

3.2 Playback phase

In playback mode, the user starts the program `mad_view`. `mad_view` parses the log file and presents the user with a list of class names for all objects recorded in the log file. Clicking on a class name displays a list of instances for that particular class (Figure 1). Instances that crashed are identified with the string “crashed” appended to their name (Figure 2).

Figure 1 **mad_view at a glance**



Note that there may be more than one instance of a class during the execution of an application. For example, in Figure 1, there are six instances of the class `fibonacci_class`. Though we can use the Mentat UID to distinguish between them, we prefer to index instances of the same class by the order in which they appear in the log file. A future improvement would be to use the variable names from the program’s source code instead of the current indexing scheme.

To reproduce the execution of a specific object, the user must first click on an object instance from the instance list or type the name directly in the “Object” text field and then click on the “Playback” button. To debug

the object, the user selects an object instance and clicks on "Debug Object". MAD will then run the specified object under control of the debugger specified in the text field "Debugger".

The output of the object is displayed in the window in which `mad_view` is started unless the debugger specified handles input and output in its own window (e.g. xdbx in Figure 2).

Figure 2 **Snapshot of mad_view and xdbx**

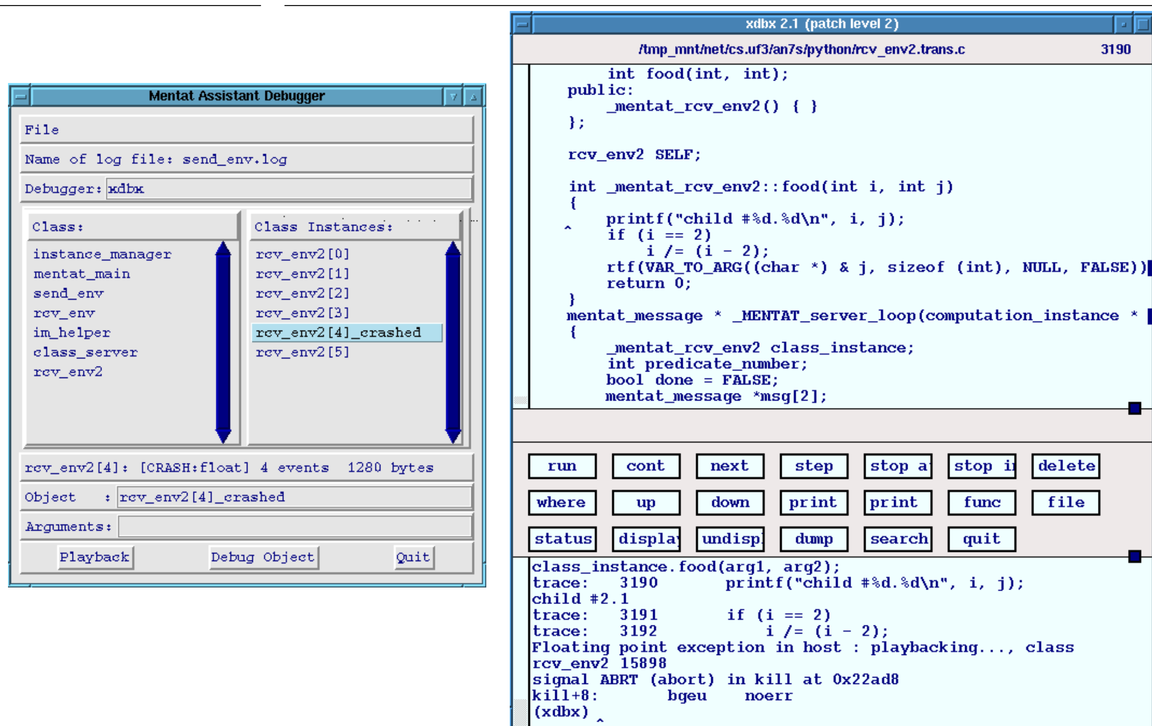


Figure 2 shows that one instance of the class `rcv_env2`, i.e. `rcv_env2[4]`, has crashed during the recording phase. In this example, the user selects `rcv_env2[4]`, and executes it with the xdbx debugger by clicking on "Debug Object". Xdbx then displays the user's source code (as transformed by the Mentat compiler).

The user can use all the features of xdbx to find the cause of the crash. In this case, the user enables tracing. As the object runs, xdbx displays each statement executed in its output window. When the object crashes, xdbx displays the offending line in its source window.

3.2.1 mad_playback

For users without access to an X-terminal, we also provide a textual interface to the playback phase: `mad_playback`. In fact, `mad_view` is simply a graphical front-end for `mad_playback`. The interface to `mad_playback` is shown below:

```
usage:  mad_playback [-h] [-f log_file] [-ddebugger] object [args...]
        debugger specifies the debugger to use
        object is the name of the object optionally indexed as:
            [i] where i is the ith instance in the log file
            [+] is the first instance that crashed
            [*] attempts to match all instances
        -h prints out a usage statement

examples:   mad_playback test_fib 5
            mad_playback -ddbx test_fib
            mad_playback -ddbx fibonacci_class[3]
```

3.3 Compiler flags needed for debugging

Two steps are necessary to provide full debugging support. The first is to keep the symbol table for the executable by setting the `-g` flag at compile time. The second is to prevent the Mentat compiler from discarding the transformed source code generated by using the `-trans` flag. Here is an example makefile:

```
MPLC = mplc
CFLAGS = -trans -g
gauss: gauss.c
    $(MPLC) $(CFLAGS) gauss.c -o $(MENTAT_USR_BIN)/gauss -lm
```

4.0 Implementation

In this section, we describe the implementation of the recording and playback phases as well as the format of the log file.

4.1 Specifying recording or playback mode

Upon startup a Mentat main program attempts to read the value of predefined Unix environment variables to determine the name of the log file (`MD_FILE`) and whether it should execute normally, in record mode or in playback mode (`MD_MODE`). In addition to these variables, `MD_OBJ` specifies the class name and instance of the object during playback.

mad_record and mad_playback are shell scripts which set these environment variables on behalf of the user.

4.2 Recording phase

To start the recording phase, the user prefixes their usual program with the command mad_record. mad_record sets MD_MODE to recording and MD_FILE to mentat_main.log (unless the user chooses a different name for the log file) before invoking the main program.

The main program transparently creates an instance of MD_recorder, a Mentat object to which all objects which comprise the application will forward a copy of their incoming messages. MD_recorder then writes the messages to a log file (Figure 3). To ensure that the resulting log file is visible to the user, i.e. on a locally mounted file system, the main program places the MD_recorder on the same host as itself and places the log file in the user's current working directory.

The main program propagates the fact that it is in recording mode to all member functions which it invokes. The invoked member functions then execute in recording mode and in turn, propagate this fact to all member functions that they invoke. The end result is that all member functions that transitively trace their invocation to the main program execute in recording mode.

The mechanism for propagating information from one member function to the next is via Mentat's environment mechanism. The invoker of a member function always sends an implicit argument — its environment[4]. Thus, an object always executes a member function within the context set by the invoker. In this case, the environment always contains the physical address of the MD_recorder object. Environments are by default invisible to users though users can access and manipulate the environment directly.

4. Mentat environments should not be confused with their Unix counterpart even though they serve a similar purpose. Unix environments allow a shell to pass information to child processes whereas Mentat environments allow Mentat objects to pass information to other objects. The value of Mentat environment variables is not restricted to strings.

When receiving a message in recording mode, a Mentat object extracts the name (i.e. the physical address) of the MD_recorder object from the environment and forwards a copy of the message to the MD_recorder object. The object then proceeds normally (Figure 3).

4.2.1 Overhead & Performance

The overhead of the recording phase consists of the amount of storage consumed by the log file and the additional network traffic generated by sending messages to the MD_recorder object. Since the log file contains a copy of all messages exchanged between Mentat objects for a given application, its total size is determined by the number and size of messages sent, with an additional fixed overhead (36 bytes) for each message. The network traffic is essentially doubled since all messages are also sent to the MD_recorder object.

The impact on performance depends on the granularity of the application and its communication pattern. However, our primary focus is not performance but rather on providing easy-to-use debugging tools in a parallel processing environment. Once the application is bug free, users can simply run their application with recording turned off.

4.2.2 The MD_recorder object

The main duty of the MD_recorder object is to record messages in a log file. Its interface is shown below:

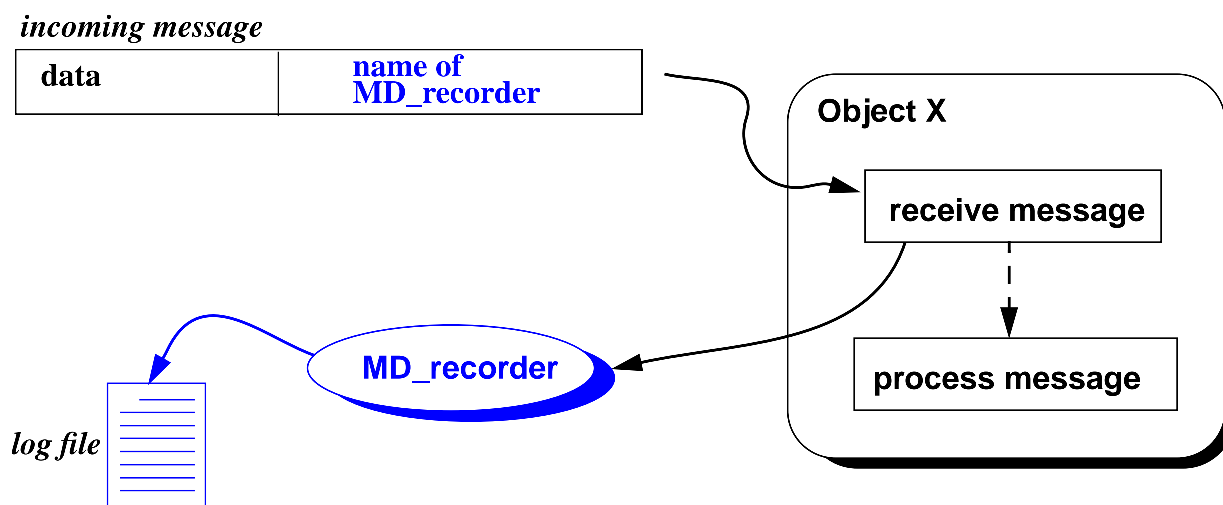
```
persistent mentat class MD_recorder {
    public:
    void create(string *app_name, int pid, int Muid);
    void record_mentat_msg(mentat_message *msg, string *cname, int Muid);
    void record_crash(string *cname, int Muid, int type);
};
```

create() is invoked once by the main program upon startup. app_name is the name of the application, pid the Unix process identifier of the main program and Muid is a unique identifier generated by Mentat for each object. create() creates the log file and names it <app_name>.log.

record_mentat_msg() records the message contained in msg in the log file. cname is the class name of the object that sent the message and Muid is its unique identifier. cname and Muid are also recorded in the log file and are used during the playback phase to distinguish objects.

`record_crash()` records the fact that an object has crashed. When an object in recording mode crashes, an attempt is made to invoke `record_crash()` with the cause of the crash contained in `type`.

Figure 3 **Forwarding of incoming messages to MD_recorder object**



Algorithm for logging messages:

- Object X receives a message
- Object X forwards incoming messages to `MD_recorder` object using the name contained in the message
- `MD_recorder` object logs messages to log file
- Object X processes the message normally

4.2.3 Format of log file

The log file consists of a sequence of event records (Figure 4). An event record contains a header and an optional variable size data region. There are five types of events defined:

```
E_MSG:          A message is logged
E_CRASH:        Object has crashed, unknown cause
E_CRASH_FLOAT:  Object has crashed due to a floating point exception
E_CRASH_MEMORY: Object has crashed due to a memory exception
E_CRASH_PIPE:   Object has crashed due to a pipe exception
```

The event `E_CRASH` is the catch all event and is used when the cause of the crash is unknown, i.e it was not due to a floating point, memory, or pipe exception.

The header contains the class name of the object that forwarded the message, its Mentat UID, the size of the data region, the event number and the event type. The data region contains the message forwarded in the case of an `E_MSG` event.

The utility for viewing log files is `mad_reader` and is included with the standard Mentat distribution. Shown below is a sample output of `mad_reader`:

```
mad_reader: logFile <mentat_main.log> filter <all> verbosity <0>
1 instances of class tmu
1 instances of class fibonacci_class
1 instances of class test_fib
1 instances of class instance_manager
```

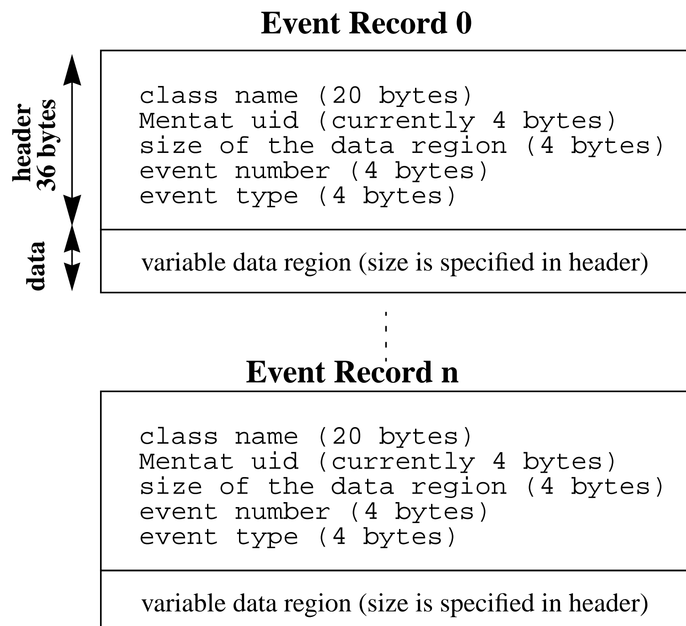
With the -l flag specified, `mad_reader` displays a more verbose output and shows the Mentat UID, the event number, the class name of the object, the size of the event data region and the event type.

```
mad_reader: logFile <mentat_main.log> filter <all> verbosity <1>
selected event: >>> [337382338] event #0 <<<: <tmu> size[336] type[MSG]
selected event: >>> [367874194] event #1 <<<: <fibonacci_class> size[336] type[MSG]
selected event: >>> [1934742628] event #2 <<<: <test_fib> size[280] type[MSG]
selected event: >>> [1621621013] event #3 <<<: <instance_manager> size[432] type[MSG]
selected event: >>> [1934742628] event #4 <<<: <test_fib> size[280] type[MSG]
selected event: >>> [1621621013] event #5 <<<: <instance_manager> size[352] type[MSG]
selected event: >>> [1934742628] event #6 <<<: <test_fib> size[280] type[MSG]
```

4.3 Playback phase

The goal of the playback phase is to reproduce the behavior of a single objects in isolation from other objects. Once this is done, programmers can debug their objects using their favorite sequential debugger. More importantly, programmers only need to pay attention to a single object at a time and can use the traditional cyclic debugging style of debugging. In other words, programmers have the tools to transform a complex task (debugging a concurrent application) into a much simpler one (debugging a single object).

Figure 4 Event records



The user starts the playback phase with the shell script `mad_playback`. While `mad_view` presents a graphical front-end, the underlying program is `mad_playback`. In addition to the Unix environment variables `MD_MODE` and `MD_FILE`, `mad_playback` also sets `MD_OBJ`. `MD_OBJ` contains the name of a Mentat class indexed with a number that describes its relative order in the log file. For example, `MD_OBJ=foo[3]`, specifies the fourth instance of class `foo`. This naming convention distinguishes between instances of the same class and is preferred over having users interact with Mentat UIDs.

An object reproduces its behavior by filtering the log file for the appropriate messages. Messages that were not originally received by the object are discarded while the remaining good messages are once again "received" by the object. The difference during playback is that messages no longer come from other objects, but instead, are extracted from the log file. Outgoing messages do not need to be sent and are discarded.

Note that playback works only when the behavior of an object is *solely* determined by the messages it receives. The user is responsible for handling the case when the object's behavior is dependent on covert channels of communications such as files, pipes or clocks.

In general, an object in playback mode will execute faster than the original version since it does not need to wait for messages to arrive but instead extracts them directly from the log file. A future enhancement would be to embed timestamps in the log file and enable users to control the rate at which objects should be played back.

5.0 Conclusion

In this report, we have presented the interface and implementation of the Mentat Assistant Debugger (MAD), a set of tools that enables Mentat programmers to debug their applications using the sequential debugger of their choice. MAD is easy to use and requires no modifications to existing Mentat code. MAD leverages off the existing base of sequential debuggers available in both the public domain and commercial sector and is designed to easily accommodate future improvements in debugging technology.

6.0 Acknowledgments

We would like to thank Mark Hyett, Lindsay Faunt and John Karpovich for their invaluable help in writing this technical report.

7.0 References

[1] Jim Gray, "Why Do Computers Stop and What Can Be Done About It?," Tandem Technical Report 85.7, June, 1985.

[2] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May, 1993.

[3] A. S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May, 1993

[4] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," *to appear in ACM TOCS*, and earlier version is available in Computer Science Technical Report, CS-93-40, University of Virginia, July, 1993.

[5] T.J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transaction on Computers*, C-36(4), pp. 471-482, April 1987.

[6] Mentat Programming Language Reference Manual, URL: http://www.cs.virginia.edu/~mentat.