# Debugging Multi-Threaded Programs with MpD

M. Krish Ponamgi
Wenwey Hseush
Gail E. Kaiser

Columbia University

## Introduction

We have developed a Multi-processor Debugger, MpD, based on *Data Path Expressions* and *Predecessor Automata*. Data Path Expressions (DPEs) is a debugger command language for the programmer to express both sequential and parallel patterns of events. Predecessor Automata (PAs) are automatically generated recognizers for DPEs. The programmer specifies DPEs to represent event patterns that may happen during program execution, and the PAs analyze the combined event streams collected from the executing threads to determine whether or not each DPE was actually matched.

DPEs specify causal dependence and independence among events executed by the same or different threads. Thus DPEs are useful for describing both desired and erroneous conditions for parallel programs whose correct behavior relies on explicit synchronization among threads (*e.g.*, message passing and locking). One example application is detecting race conditions. DPEs cannot express timing constraints and thus are not suitable for debugging real-time programs.

DPEs were inspired by the Generalized Path Expressions developed by Bruegge and Hibbard[1] for debugging sequential programs written in Ada or Pascal. DPEs are essentially regular expressions over program events, but with a concurrency operator, indicating causal independence, added to the regular expression notation. Throughout this paper, when we say two events are concurrent, we mean they are causally independent, but not necessarily actually simultaneous.

PAs are analogous to Finite State Automata (FSAs), which recognize regular expressions. But in PAs, each transition is labeled with both a program event and its set of immediate predecessor events, on which the program event is causally dependent. Thus while Finite State Automata can recognize only totally ordered streams of events, PAs can recognize partially ordered streams.

We have implemented a concurrent debugger, called MpD for Multi-processor Debugger, that allows the programmer to specify interesting patterns of events in terms of DPEs. MpD automatically generates corresponding PAs, and recognizes whether or not each DPE is matched during the execution of the concurrent program. Our system is written on top of a version of gdb[*] developed by Caswell and Black[2] to run on CMU's Mach operating system.

MpD can be used to debug C programs written using the Cthreads package[3] running on Mach 2.2 on IBM's experimental 8CE multiprocessor[4]. An 8CE machine consists of up to eight processing elements, each with an RT 125 CPU and 8MB local memory, and a 16MB global memory shared among the processing elements. It runs as a co-processor on a regular IBM PC/RT 125, which supports I/O. The full set of commands available within MpD's dpe command shell are listed in the Box. This shell is entered from within gdb by typing the *dpe* command. Figure 1 illustrates the structure of the MpD system. MpD consists of about 5,600 lines of lex and yacc rules and C code.

**Figure 1:** Structure of MpD System

The Caswell-Black version of gdb includes the kernel facilities needed to deal with multi-threaded programs. This version supports setting breakpoints for individual threads, as opposed to every thread that attempts to execute a particular statement, and can control the execution of

[*]The original gdb debugger is distributed in source code form by the Free Software Foundation.

individual threads. Throughout this paper, when we say "gdb", we mean this Caswell-Black version of gdb. MpD is built on gdb's low-level layer to provide high-level support for describing patterns of events with Data Path Expressions, where the DPEs specify relationships among the events in different threads, and for recognizing such patterns with PAs.

In our first paper on concurrent debugging[5], we introduced the idea of Data Path Expressions and gave some examples of how we imagined they might be useful for concurrent debugging. In another previous paper[6], we characterized the modeling power of a hierarchy of syntactically restricted DPE languages in terms of a hierarchy of Extended Petri Net models. We introduced Predecessor Automata, and formally described the algorithm to translate DPEs to PAs. We have also proved the equivalence in modeling power of PAs and the subclass of DPEs presented here[7]. In this paper, we present MpD for the first time, discuss actual implementation issues, and describe a real example of using MpD to debug a concurrent application program.

## Idealized Data Path Expression Language

A DPE consists of two components: an expression part and an action part. Expressions consist of events and operators that specify the behavior of program execution. Actions are performed by the debugger when the particular behavior is recognized during execution.

A simple event is an assertion that a particular point in the program has been reached, for example, a particular subroutine has been called ("S" means that the `S` subroutine has been called). More complex events may include conditions, such as a particular variable address is being accessed at that point ("Read[X]" means that the `Read` subroutine has been called with some variable `X` as its argument). The kinds of conditions that can be expressed depend on the implementation. A set of operators — sequencing (;), exclusive selection (+), repetition (*) and concurrency (&) — express the basic relationships among events. The meanings of these operators are summarized in Table 1. Parentheses are used for grouping events and operators. Actions are stated between braces "{...}". The kinds of actions that can be expressed depend on the implementation.

For example, the DPE shown in Figure 2 (a) specifies that either a `Write` is causally dependent on a `Read`, or vice versa. Further, the `Read` and `Write` events are conditional on both accessing the same object (some `X`). In contrast, Figure 2 (b) specifies that the `Read` and `Write`, on the same object `X`, are causally independent. The detection during execution of this

| Expression | Description |
|---|---|
| **A;B** | **A** causally precedes **B**. |
| **A+B** | Either **A** or **B** occurs, but not both. |
| **A*** | $\varepsilon$ **+ A + A;A + A;A;A + ...** |
| **A&B** | **A** and **B** occur causally independently. |

**Table 1:** Operators

```
((Read[X] ; Write[X]) + (Write[X] ; Read[X]))  # expression
{echo "Sequential Access Read and Write"}      # action
```

(a)

```
(Read[X] & Write[X])
{echo "Concurrent Access Read & Write"}
```

(b)

**Figure 2:** Example DPEs

race condition might indicate a programming error. Each example includes an action to print a message reminding the user as to the meaning of the matched DPE.

When the same event name appears multiple times in a DPE, it refers to a different instance of the event. Thus "A*" means zero or more sequential occurrences of event A, and "A & A" means two concurrent executions of distinct events named A. This means that the DPE "( A ; S ; B ) & ( C ; S ; D )" describes two separate events S rather than synchronization between the two concurrent sequences of events. To describe synchronization between the two concurrent sequences, the synchronization event is prefixed with "$", as in "( A ; $S ; B ) & ( C ; $S ; D )".

## MpD Data Path Expression Language

In the MpD implementation of DPEs, an event is defined by a breakpoint inserted into the program execution image using gdb’s normal facilities. Therefore numerical breakpoint labels rather than mnemonic event names are used. The DPEs are then constructed as expressions on breakpoint numbers using the operators presented above. Event conditions are written in terms of gdb actions, for example to save variable addresses, attached to individual breakpoints. Actions may involve artificial variables (debugger as opposed to program variables) and I/O. MpD automatically inserts actions at all breakpoints to force gdb to invoke MpD when a breakpoint is reached. When invoked at a breakpoint, MpD indicates to gdb whether or not some component of a DPE is matched, and the event condition actions are performed only if the breakpoint matches part of a DPE. Actions may also be specified to be carried out only after an entire DPE containing a particular breakpoint is recognized. This might be used to compare the variable addresses recorded at different breakpoints.

```
(gdb) set $read_var = 0
(gdb) set $write_var = 1
(gdb) actions
event Read {
set $read_var = X
if_concurrent
  if ( $read_var == $write_var )
    echo "Concurrent Access Read & Write"
}
event Write {
set $write_var = X
if_concurrent
  if ( $read_var == $write_var )
    echo "Concurrent Access Read & Write"
}
end
Breakpoint 10 0x123: file Demo.c, line 37
Breakpoint 11 0x456: file Demo.c, line 99
(gdb) dpe
DPE> dpe 10 & 11
DPE> quit
(gdb) run
(gdb) state
```

**Figure 3:** Defining MpD DPE

Say the user wished to define the DPE shown in Figure 2 (b). In Figure 3, the **bold** font indicates MpD/gdb prompts and the `typewriter` font indicates user input. The user follows the steps listed in Figure 4.

## Predecessor Automata Recognizers

Event streams are recognized during execution by PAs generated from the active DPEs. Zero or more DPEs can be active, in which case MpD will inform the programmer as to which DPEs were matched, if any, during the previous program execution under MpD.

1. Initialize two gdb artificial variables for later use, `$read_var` and `$write_var`.
2. Use the gdb *actions* command to enter a shell for defining events and corresponding actions.
3. Use the *event* command (an alias for gdb's `break` command) to define the event of entering the `Read` function.
4. Associate an action with the individual `Read` event to save the address of the function argument.
5. Use the *if_concurrent* command we added to gdb to associate an additional action, to compare with another saved address, to be performed whenever any entire DPE containing the `Read` event is matched.
6. Repeat the above three steps for the `Write` function.
7. End the gdb actions shell, which causes gdb to print the numbered labels assigned to the two breakpoints.
8. Use our *dpe* command to enter the dpe command shell.
9. Construct the "10 & 11" DPE (which implicitly constructs the corresponding PA).
10. Quit the dpe shell and run the program.
11. Use our *state* command to request information about whether or not the DPE was matched. MpD organizes the events that occurred by thread and indicates their ordering (not shown).

**Figure 4:** Steps for Defining and Using DPEs

PAs are analogous to finite state automata, but PAs recognize concurrent partially ordered event streams expressible as DPEs while FSAs recognize only sequential totally ordered event streams expressible as regular expressions. Note that all regular expressions are also DPEs, but those DPEs containing any concurrency operators are not regular expressions.

An FSA makes a transition from one state to another when its input event matches the event labeling that transition. Certain states are designated as final states, and if the input ends when the FSA is in a final state, or a final state can be reached by null transitions, the corresponding regular expression has been matched by the input event stream, otherwise not.

Similarly, a PA makes a transition from one state to another when its input event plus an encoding of the set of events causally preceding the current event together match the <event, predecessor-set> pair labeling that transition. Note that only the immediately preceding events in each thread of control is contained in the predecessor set, not the full history of events. There are multiple predecessors only when the event reflects a synchronization among multiple threads. As with FSAs, there are final states, and the partially ordered event stream has been matched if and only if the PA is in a final state when the input ends, or a final state can be reached via null transitions.

PAs are automatically constructed by MpD as the corresponding DPEs are input by the user. The parse tree is created by a DPE parser generated using the yacc tool. Then a postorder traversal of the parse tree constructs the PA representation.

The postorder traversal creates a left PA and a right PA for each internal node, which are then *fused* according to the operator at the internal node. For ";" the two child PAs are concatenated, for "+" the union is taken, and for "*" a loop as well as a null transition are added, all the same as in construction of an FSA from a regular expression. For "&", the concurrency operator, the two PAs are associated with each other but not directly combined. Instead, the run-time component of a PA recognizer keeps track that <u>all</u> the ‘‘sub-PAs’’ fused together with a concurrency operator are completed before entering the final state.

Figure 5 illustrates graphically the two PAs generated to recognize the DPEs of Figure 2. Each transition in a PA is labeled with a pair, consisting of the event name and the set of immediate predecessors. If an event has no immediate predecessors, then the second part of the pair is an empty set, represented as ".".

## Run-Time Support

When events are described as concurrent in a DPE, the PA recognizer must check two conditions. The first is that the events occur on different threads, since of course there is causal dependency between the events on the same thread. This check does not suffice, however, since there may be causal dependency between events in different threads due to synchronization between their threads.

The construction of predecessor automata is independent of the particular concurrent programming language used. However, run-time support is needed for PA recognizers to exploit the execution context to determine predecessor relations among events. This support is inherently specific to the synchronization primitives used for concurrent programming. In our case, this is the routines provided by the Cthreads library.

(a)

(b)

**Figure 5:** Example PAs

When programming with the Cthreads package, synchronization may be done using `mutex_lock` and `mutex_unlock`. A thread calls `mutex_lock` to lock a resource, forcing other threads to wait until it calls `mutex_unlock`. If the locking mechanisms in an application are not properly structured, then multiple threads may execute in parallel on the same resource. This often results in undesirable race conditions.

MpD detects this situation by identifying which `mutex_lock`s in different threads are competing with each other. A special run-time table is created for each thread. Each time a `mutex_lock` is attempted and a thread waits on a resource, the thread that currently owns that resource is identified. Any `mutex_lock` waits of the owner thread are inherited by the newly waiting thread. Thus the predecessor table can be accessed to immediately identify whether two events are concurrent or not.

Synchronization implemented by the Cthreads `condition_wait` and `condition_signal` primitives also induce causal dependency. If one thread waits for another due to a `condition_wait` (directly or indirectly through another thread), the concurrent relationship does not hold. MpD informs the user appropriately.

In a similar fashion, the PA recognizer enforces dynamic checks for sequential relationships. One of two conditions must be satisfied:

1. The event and its predecessor are in the same thread.
2. The event and its predecessor are in different threads, but serialized via the locking or signaling primitives.

In every other execution sequence, the events are understood as concurrent.

For the DPE "( A & B ) ; C", event `C` has as its predecessors both events `A` and `B`. For all of the predecessors, MpD checks to see whether the sequential relationship holds. Similarly for the DPE "A ; ( B & C )", MpD must ensure that there is a causal dependency between both `A` and `B` and between `A` and `C`.

## Additional Support for Cthreads

Along with the DPE support for multi-thread programming, specific facilities are included in MpD to aid Cthreads programmers in the Mach environment. (We could similarly have included specific support for other programming libraries.)

The most useful Cthreads-specific support included in MpD is automatic detection of deadlocks caused by an improper sequence of `mutex_lock`s. Each time a thread attempts to lock a resource using `mutex_lock`, MpD's internal list of what resources each thread holds/wants is updated. As this list is updated, a graph of dependencies is created. If the graph forms a cycle leading back to the current thread, the user is warned of the impending deadlock with a line by line printout of the culprits.

The user can monitor Mach ports from which messages are sent. Each time a message is sent to a port that is being monitored, MpD logs which line in the source code sent the message, i.e., it identifies which `msg_send` was responsible. This is useful for quickly checking the origin of non-sensical messages.

MpD does numerous checks for common Cthreads programming errors. For example, the `cthread_init` routine <u>must</u> be called prior to using any of the other Cthreads routines. Failure to do so results in corruption of memory and core dumps. MpD checks whether this function is called prior to any other Cthreads function. A Cthread, once created, can be joined (via `cthread_join`) or detached (via `cthread_detach`) only once during execution. This is checked by monitoring all Cthreads to make sure only one or the other is done. Furthermore, any Cthread attempting to join itself (via `cthread_join`) results in deadlock. MpD detects this condition and warns the user.

## Using MpD

The Parallel Logic Simulator consists of about 1,900 lines of C code. The application ran correctly on the 8CE's RT front end using the co-routines implementation of the Cthreads library (libco_threads.a). However, using the real parallel implementation of the Cthreads package on the 8CE co-processor (libthreads.a), the application would run briefly and then suspend itself in some unknown state. This occurred every time the application was tested. Note that our approach to debugging here assumes reproducibility.

Since we were debugging unfamiliar code, our first step was to determine how far the application got before hanging. We used regular gdb breakpoints to determine that the application reached the `setup` routine in the setupMach.c file (this is one of the application program files, not part of the Mach implementation).

Since we knew that the application worked using co-routines but not using real parallelism, our next step was to determine how many threads were being created by the application. So we defined the action "$i = $i + 1" on entrance to `setup`, using normal gdb facilities as shown in Figure 6. "$i" is a gdb debugger variable rather than a program variable. This allowed us to count the separate invocations of the `setup` function.

```
(gdb) set $i=0       # dummy counter to count number of threads
(gdb) actions
event setup {
  # setup is the name of forked routine (in setupMach.c)
$i = $i + 1
cont
}
end
```

**Figure 6:** Adding Counting Action to `setup` Breakpoint

The number of threads being forked was determined by printing the value of `$i`, which was 4.

A single-thread debugger would not be able to take even this simple step of counting the number of concurrent threads.

Within `setup`, the C library (libc.a) routines `fopen` and `printf` were called, as shown in Figure 7. However, our Mach Cthreads manual recommends that functions in the C library should never be called without `mutex_lock`/`mutex_unlock` pairs protecting them. This is because the version of the C library that we used does not support concurrent access, since the code is not reentrant.

Thus, since there were four concurrent executions of `setup`, we suspected that these C library functions might be executed concurrently. If so, that might be the cause of the program hanging.

```
if (!(dbgf = fopen (dbgName, "w"))){
  /* line 298 of setupMach.c */
    printf("[dbg] %s :", dbgName);
    quit("cannot open debug file");
    }
```

**Figure 7:** Suspicious Portion of `setup` Code

To detect whether `fopen` was indeed being called by multiple threads concurrently, we defined a breakpoint on reaching the statement that calls `fopen`. gdb assigned the breakpoint number 1. This is illustrated in Figure 8.

```
event setupMACH.c:298 {
  # line calling fopen and prior to calling printf
cont
}
end
```

**Figure 8:** Setting Breakpoint at Suspicious Statement

We then defined the DPE "1 & 1 & 1 & 1", representing four concurrent threads executing this same statement causally independently. The program was executed, and this DPE was confirmed. Thus, `fopen` (as well as `printf`) was being invoked by four threads concurrently. This bug would have been awkward to detect using a single-thread debugger.

We modified the application program to protect all calls to functions in the C library with `mutex_lock`s/`mutex_unlock`s. The program was recompiled and reexecuted, and execution proceeded normally with correct results. Thus MpD helped us to find and repair the bug.

## gdb-related Implementation Issues

Our first major difficulty in implementing MpD was getting gdb working on the 8CE. We found some problems with the symbol table format produced by the hc2 compiler for the RT. This symbol table format could not be understood by gdb. Since we were unable to modify hc2, we modified gdb instead to work with hc2.

As discussed previously, each MpD event is represented by the numerical label of a breakpoint installed using the underlying gdb facilities. The event occurs when the breakpoint is reached in some thread's execution of the corresponding code. Although the user necessarily sets the breakpoints using gdb, from his or her perspective MpD does not actually stop program execution at these breakpoints. What actually happens is that gdb suspends the program briefly and indicates to the MpD recognizer that the breakpoint was reached. MpD processes this breakpoint and returns an indication to gdb as to whether it matched a transition in an active DPE. If MpD returns successfully, any actions associated with that breakpoint are executed by gdb. Execution is then continued. This is all transparent to the user.

In the original gdb, breakpoints and actions could not be created in one continuous flow. Instead, the gdb user had to insert breakpoints first, and then attach actions individually to each breakpoint. We modified the gdb *actions* command to allow breakpoints and their associated actions to be described at the same time. Now the user simply types "actions" to enter the actions definition shell, describes the location of the breakpoint normally, and writes the desired actions enclosed within braces ({...}). This was illustrated in the various breakpoint examples in the previous sections. This made gdb actions easier to use, but it would have been better to have had a more powerful action language, such as provided by the Dalek[8] extension of gdb.

## 8CE-related Implementation Issues

A bug in the 8CE hardware presented a significant obstacle to implementation of MpD. gdb was originally unable to single step in any thread of execution, both for single- and multi-threaded applications. The reason for this was that the kernel on the 8CE was failing to single instruction step any application.

The single instruction step is crucial for breakpoints. The mechanism by which breakpoints are implemented depends on this feature. Each time a breakpoint is inserted, the actual machine code is modified and a special machine instruction replaces it. This instruction, when encountered during execution, generates a software trap at which point the debugger gains control.

To continue after this point, the original contents of the machine code is put back and a single instruction step is done to proceed past this point. After this single instruction is finished, the breakpoint instruction is reinserted and execution continues normally. The motive for this double-stop is that the breakpoint encountered after the first breakpoint may be this very same breakpoint (the breakpoint could be in a loop, goto, recursive call, etc.). This necessitates that a breakpoint be replaced immediately after its removal.

The failure of the 8CE to single instruction step caused gdb to lose control after the very first breakpoint. The software-trap that was never generated manifested itself as a message that timed out. Investigating the problem led us to the following comment in the kernel source code:

```
/* simulated -- unreliable in hardware */
```

The simulation represented the bit that is set when a single instruction step is needed.

After discovering this fatal error, we implemented a patch. Our idea was to generate for every breakpoint, two adjacent breakpoints. When the first breakpoint is encountered, it is removed and the second is inserted. After the second is reached, it is removed and the original is replaced. This ‘‘jitter-step’’ simulates the single instruction step. Our solution has proven to be very effective in overcoming the hardware bug.

## Possible Improvements

There are several improvements that could be made to the current MpD system. The most obvious deficiency is the user interface. For example, we could modify gdb to support mnemonic event names rather than numbers, but we have tried to minimize our changes to gdb. Bates[9] has developed a concurrent debugger that uses mnemonic labels, but his system requires modifications to the source code being debugged, in order to insert subroutine calls that produce events corresponding to these labels during execution. In contrast, our technique does not require any modifications at all to the source code, and relies on gdb’s existing ability to insert breakpoints into the execution image. Individual breakpoints can be added or deleted and their actions can be changed at any time, making it easier to construct DPEs. It would of course be better if operating system support or hardware were available[10], so invasive techniques could be avoided entirely.

Another improvement would be to reduce DPE expressions to minimize backtracking. The effort would be similar to writing a regular expression reducer. The end result would be that the PAs would be more compact, making recognition more efficient.

Some of the MpD code is machine dependent, particularly to examine the parameters of Cthreads library functions (`cthread_fork`, `mutex_lock`, `cthread_init`, etc.). The parameters are placed only in specific locations on the stack and in certain registers, and access to this information is likely to be different for different machines. To make MpD portable, it would be necessary to write a library interface package that enables MpD to read the necessary function parameters and then call the appropriate function in a machine-independent way.

Probably the most significant improvement would be to reimplement the interface between gdb and MpD. MpD is essentially a subroutine within gdb, but we now realize it should have been implemented as a separate thread. Further, our current implementation of MpD supports only a single process (containing an arbitrary number of threads). gdb runs as a separate process on the designated master processor and controls the application program as an inferior process. To scale up to large applications that themselves contain multiple processes, a separate instance of gdb would then have to be attached to each process. The event information collected by each gdb would be passed to a centralized MpD recognizer, which itself could be parallelized into multiple threads. This is an interesting area for future research.

## Conclusions

The primary contributions of this paper are to show how our Data Path Expression debugger command language and the corresponding Predecessor Automata recognizer mechanism can support multi-threads debugging. This support is useful for detecting errors involving race conditions and deadlock, and in general can aid programmers in understanding concurrent program behavior. We have implemented MpD on top of the Caswell-Black version of gdb for Mach, to debug programs written in C using the Cthreads package, and it runs on the IBM experimental 8CE multiprocessor. We described the difficulties we encountered during the implementation, and explained how we overcame these obstacles to develop a practical multiprocessor debugger. MpD has so far been used to debug two externally developed applications, one of which is described here.

## Acknowledgments

The implementation of MpD on 8CE was undertaken as part of a joint study with Dr. Colin G. Harrison of the IBM T.J. Watson Research Center. Part of this work was done while Hseush was a summer employee at IBM in 1989, part while Ponamgi was a co-op student employed by IBM in 1989-90, and part while Kaiser was an Academic Visitor at IBM in 1990. We would like to thank Krish Kannan, Doug Kimelman, P.R. Kumar, Chuck Marvin and Charlie Perkins for their suggestions and technical expertise. Travis Winfrey also provided some systems support for the 8CE located at Columbia. The parallel logic simulator application was supplied by Robert Mueller-Thuns of the University of Illinois at Urbana-Champaign.

Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, DEC, IBM, SRA, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research. Hseush is supported in part by the Center for Telecommunications Research.

## References

1. Bernd Bruegge and Peter Hibbard, ‘‘Generalized Path Expressions: A High-Level Debugging Mechanism’’, *The Journal of Systems and Software*, Vol. 3, No. 4, December 1983, pp. 265-276.

2. Deborah Caswell and David Black, ‘‘Implementing a Mach Debugger for Multithreaded Applications’’, *USENIX Conference Proceedings*, Washington DC, January 1990, pp. 25-39.

3. Eric C. Cooper and Richard P. Draves, ‘‘C Threads’’, Tech. report CMU-CS-88-154, Carnegie Mellon University Department of Computer Science, June 1988.

4. Armando Garcia, David J. Foster and Richard F. Freitas, ‘‘The Advanced Computing Environment Multiprocessor Workstation’’, Tech. report RC14491, IBM Research Division, T.J. Watson Research Center, February 1989.

5. Wenwey Hseush and Gail E. Kaiser, ‘‘Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language’’, *ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, Madison WI, May 1988, pp. 236-246, Special issue of *SIGPLAN Notices*, 24(1), January 1989.

6. Wenwey Hseush and Gail E. Kaiser, ‘‘Modeling Concurrency in Parallel Debugging’’, *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*,

Seattle WA, March 1990, pp. 11-20, Special issue of *SIGPLAN Notices*, 25(3), March 1990.

7. Wenwey Hseush, Timothy S. Balraj and Gail E. Kaiser, ‘‘Modeling Regular and Safe Concurrent Processes’’, Tech. report CUCS-039-90, Columbia University Department of Computer Science, July 1990.

8. Ronald A. Olsson, Richard H. Crawford and W. Wilson Ho, ‘‘Dalek: A GNU, Improved Programmabel Debugger’’, *Summer 1990 USENIX Conference*, USENIX Association, Anaheim CA, June 1990, pp. 221-231.

9. Peter Bates, ‘‘Distributed Debugging Tools for Heterogeneous Distributed Systems’’, *8th International Conference on Distributed Computing Systems*, IEEE Computer Society, San Jose CA, June 1988, pp. 308-315.

10. Charles E. McDowell and David P. Helmbold, ‘‘Debugging Concurrent Programs’’, *ACM Computing Surveys*, Vol. 21, No. 4, December 1989, pp. 593-622.

### Box: MpD User Commands

MpD provides a command-line user interface. All the *help* messages provided by MpD for these commands are listed below, separated by bars. MpD is invoked from within gdb, by entering the *dpe* command to gdb.

```
Usage:  help
        help <commands>
Effect:
The word "help" by itself displays this file.  Typing
help  followed by a specific command gives help on usage
and effect  of the command. Available commands are:
dpe -- Enters dpe into dpe table and creates pa
(predecessor automata) for it.
display -- Displays the dpe table or specific dpe
statistics.
quit -- Exits dpe mode and re-enters gdb mode.
delete -- Deletes specified dpe(s) from dpe table.
active -- Makes specified dpe from dpe table active for
execution modeling.
renum -- Renumbers entries in dpe table from 1 upwards.
skipon -- Don't stop at mutex lock/unlock specified.
skipoff -- Stop at mutex lock/unlock if encountered.
receive -- Receive messages from a specified port.
ignore -- Ignore messages from a specified port.
help -- Displays this file.
```

---

```
Usage:  active <number>
Effect:
Current active dpe for execution modeling becomes entry
specified by number in the dpe table.  Old active dpe
switches places with new active dpe in the dpe table.  The
dpe in entry 1 in the dpe table is ALWAYS the currently
active one.  To see the dpe table use command display.
```

---

```
Usage:  delete <number(s)>
Effect:
Deletes dpe(s) corresponding to the number(s) in the
dpe table.
Example: delete 4 5 7    This will delete dpes 4, 5 and 7.
```

---

```
Usage:  display
        display <number>
```

```
        display locks
        display ports
Effect:
Displays the dpe table.  If given a specific entry
in dpe table, detailed statistics about the dpe are
printed.  With the key word "locks" it displays current
list of mutex lock/unlock variables being ignored.  The
key word "ports" lists currently monitored ports to
appear.
```

---

```
Usage:  dpe <dpe expression>
Effect:
Creates predecessor automata (pa) out of specified dpe.
A data path expression (dpe) tries to model the execution
flow of the breakpoints set in gdb.  To specify the
execution flow the operators used are:
;  (sequential)
+  (exclusive or)
*  (repetition)
&  (concurrent)
Examples of dpes modeling execution flow would be:
dpe 1;2  means breakpoint 1 is seen then breakpoint 2.
dpe 1&2  means breakpoint 1 is concurrently with
  breakpoint 2.
dpe (1;2)*;(3+4);5
        means breakpoint 1 followed by breakpoint 2 can
        occur 0 to any number of times, then either
        breakpoint 3 or 4 occurs, followed by breakpoint 5.
The operators may be grouped by parentheses to any depth.
In addition, several dpes may co-exist but ONLY ONE is
active (i.e. gdb events are compared to it) at any given
time.  The dpes are placed into a "dpe table" which can be
manipulated by various commands while in the dpe parser
(see help for list of available commands).
When the program executes in gdb the actual order of the
breakpoints occurring is compared with the specified dpe.
At the end of execution the command "state" provides
information comparing the dpe model against actual
execution.
```

---

```
Usage:  ignore <port>
Effect:
All messages sent to specified port via the msg_send()
function will be ignored when we receive_msg().  This is
usually used to turn off a port the receive command
```

**specified earlier.**
**Example: ignore port1**
**(port1 is defined in the program to be a port)**

---

**Usage: quit**
**Effect: User exits dpe manipulation mode and re-enters gdb mode.**

---

**Usage: receive <port>**
**Effect:**
**All messages sent to specified port via the msg_send() function will be printed out when we receive_msg(). The sender is identified by the line of code which called msg_send().**
**Example: receive port1**
**(port1 is defined in the program to be a port)**

---

**Usage: renum**
**Effect:**
**Renumbers dpes in dpe table assigning them the numbers 1 through # entries in table. Useful after multiple deletions and swapping.**

---

**Usage: skipoff <locked_variable>**
**Effect:**
**If this variable was turned off by a previous skipon, this command turns it back on. Each time we encounter the locked variable in either mutex_lock() or mutex_unlocked(), we pass it to the recognizer.**
**Example: skipoff pr_lock**
**(pr_lock is defined in the program to be a lock variable)**

---

**Usage: skipon <locked_variable>**
**Effect:**
**If mutex lock/unlock breakpoints (with the "mpd mutex" command under gdb) are inserted, every call to mutex_lock() and mutex_unlocked() are passed to the recognizer. With the command skipon one can specify which ones NOT to pass (skipon = skip passing on locked_variable). This is useful,**

**for example, if we want to skip passing on I/O lock variables. If more than 15 such entries are needed, a hard-coded constant needs to be changed.**
**This also allows you to skip various types of error checking:**

```
    skipon warnings     /* skip mutex_lock() error
                           messages */
    skipon deadlocks    /* skip deadlock checking */
```

**Example: skipon pr_lock**
**(pr_lock is defined in the program to be a lock variable)**