Tracking the Elusive Mandelbrot Set Error Using Event Based Behavior Abstraction

Peter C. Bates

COINS Technical Report 89–06
February 1, 1989

Laboratory for Cooperative Parallel and Distributed Computing
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332.

Contents

1. Introduction **1**

2. The Concurrent Mandelbrot Set Generator Program **2**
- 2.1 Excruciating Detail 4
  - 2.1.1 Shared Data Structures and Their Locks 4
  - 2.1.2 Initial Process Task 6
  - 2.1.3 Worker task 6
- 2.2 Instrumentation for Behavioral Abstraction 9
- 2.3 All This Precision and Death Was Swift 11

3. The First Pass – It's a Non-Terminating Critical Section **13**
- 3.1 Implementation and Use of Critical Sections 13
- 3.2 Instrumentation of the Mutual Exclusion Lock Routines 14
- 3.3 Simple Investigation of Critical Section Behavior 15

4. X Window Interface as the Defendant **18**
- 4.1 X Windows Instrumentation 18
- 4.2 Looking Through the X Windows Viewpoint 19
- 4.3 The Fix 24

5. Retrospective on the Instrumentation **25**
- 5.1 Probe Policy 26
- 5.2 Probe Effects 29
- 5.3 Mechanism 30

6. Conclusions **33**

List of Figures

1	The Mandelbrot set display window	4
2	Important Shared Data	5
3	Work unit allocation routine	5
4	Initial Process Task	7
5	worker task main loop	8
6	Store Patch routine	10
7	Failing event stream for 4 workers	12
8	Model for correct high-level Patch filling	14
9	Instrumented Library routine	15
10	LockPair high-level event	16
11	Summary of LockPair investigation	17
12	`PatchStored` high-level model	19
13	Results of `PatchStored` high-level model	20
14	*Outlines for XStorePixmapZ()* and *XPending()*	22
15	Telltale high-level event instances	23
16	*D_Shuffle* model for initial task/worker interaction	24
17	Mandelbrot Set Generator component layering	34

Tracking the Elusive Mandelbrot Set Error Using Event Based Behavior Abstraction

Peter C. Bates

February 1, 1989

Laboratory for Cooperative Parallel and Distributed Computing
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

Mandelbrot sets delight thousands of Scientific American readers and multitudes of computer hackers that enjoy the scenic beauty created by the boring regularity of a simple algorithm. A program that creates Mandelbrot sets using a parallel algorithm was developed to exercise the parallelism of the Sequent Symmetry Multiprocessor and the display capabilities of the DEC VAXStation II/GPX.

The program was converted to be used as a canonical example of an event generator for Event Based Behavioral Abstraction toolset tests. The program was a fine program that due to a particular feature of its output display caused severe performance degradation on the remote VAXStation II that was only used to display the Mandelbrot set results. An attempt to remedy that performance problem involved removal of the effects of a single procedure call, that resulted in consistent but not determinate failure of the Mandelbrot set generator program.

This paper is a detailed report of the effort required to isolate and explain the error using only the Event Based Behavior Abstraction toolset as the investigative tool.

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332.

1. Introduction

A Mandelbrot set [4,6] is created by an iterative process over regions of the complex number plane. Typically, to investigate a Mandelbrot set, the iterative process is applied to a region of the complex plane over which it makes sense to evaluate it and the results are displayed on a computer graphics display. The display of the generated set will be characterized by a color coding of the iterations to divergence of each point that is selected for evalutation–sometimes resulting in quite spectacular displays.

The algorithm that generates a Mandelbrot set is inherently parallel, all points to be investigated are independent of each other. Maximum parallelism could be achieved if a processor element could be assigned to each point selected from the complex plane.

The Laboratory for Cooperative Parallel and Distributed processing acquired two Sequent Symmetry multiprocessors to serve as the basis for several large research projects involving cooperative distributed problem solving and parallel computation. A program to create Mandelbrot sets using a parallel algorithm was written to:

1. Serve as a simple performance testing benchmark,
2. Demonstrate the combined use of the multiprocessor as a powerful compute server and graphics workstations connected by local area network as a rendering device for computed results,
3. Serve as a means to evaluate several available synchronization techniques, and
4. To test several types of network communication software.

The program performed well and met the attained all of the above goals.

Event Based Behavior Abstraction (*EBBA*)[3,1,2] is a high-level debugging approach which treats debugging as a process of creating models of actual behavior from the activity of a system and comparing these with models of expected system behavior. A toolset has been created that permits investigation of programs using the techniques developed by this approach. Canonical examples of distributed programs have been used to debug the toolset programs and to refine the algorithms that constitute *EBBA*. In order to allow the toolset to be used with a new system a small set of low-level routines that deal with event generation and debugging tool interaction must be ported to the new system and made available to programs that need debugging services. The concurrent Mandelbrot set generator was chosen as an example of a parallel program to serve as a testbed for the *EBBA* low-level machinery on the Sequent Symmetry/Dynix systems.

The hardware and software environment in which this experiment has taken place contains the following configuration:

1. Sequent Symmetry shared-memory multiprocessor containing 12 Intel i386 processors and 96 Megabytes of physical memory. There are no floating point accelerators on any of the

processors. The operating system is Dynix, the Sequent version of Berkeley UNIX extended to emulate the System V interface and deal with concurrent shared memory programs.

2. DEC VAXstation II/GPX workstation with MicroVAX II cpu, 9 megabytes of physical memory, and the 8 plane color graphics controller. The operating system is ULTRIX, the DEC version of Berkeley UNIX, using the X Windows System developed by MIT Project Athena.

These systems are linked via Ethernet along with approximately 150 other workstations and minicomputers executing a variety of systems.

Due to a particular feature of the MSG output display, the program was responsible for a considerable performance degradation on the VAXstation used to display the results. A seemingly innocuous program change to alleviate this performance problem caused the program to fail. The failure itself was not readily reproduced due to the true concurrency and resulting non-deterministic behavior, but was somewhat consistent.

The search for the error was undertaken using only the *EBBA* toolset. The resulting trail is a good demonstration of the use of the *EBBA* approach and is reported here as it happened. The next section describes the MSG program in detail and its initial instrumentation for behavior abstraction. The third section explains the first pass at looking for the error as a simple case of livelock–an obvious first choice, and why the toolset showed that this was not exactly the problem. Section four continues and describes where the error was found. Following that is a section (five) that provides a retrospective on program instrumentation. The last section draws some conclusions from the experiences obtained in tracking the Great Elusive Mandelbrot Set Error.

2. The Concurrent Mandelbrot Set Generator Program

The concurrent Mandelbrot Set Generator (MSG) is a program that generates and displays portions of the Mandelbrot Set using a parallel algorithm. The Mandelbrot Set is located in the complex number plane where each point is represented by a number of the form $a + bi$. Without difficulty, a and b may be treated as coordinates of the point. The function (M) computed by the MSG program is:

$$z = z^2 + c$$

where z and c are complex numbers. In order to compute a Mandelbrot Set and display the results a square region of the complex number plane is selected and mapped onto a display window. The real part (a) of each z will be located on the horizontal axis (x) of the window; the imaginary part (bi) on the vertical (y) axis.

The region to be investigated is specified by selecting an origin ($min(x, y)$) and the length of a side (in the plane). For example the region bounded by $-0.19 \leq x \leq -0.13$ and $1.01 \leq y \leq 1.07$ would be specified as $x = -0.19, y = 1.01$ with a side $length = 0.06$. Mapping this region onto a

display window will fix the number of candidate points to be tested by M. Each point corresponds to a single pixel in the display window.

The set is generated by setting c to be a point in the selected region and setting $z = 0$. M is iterated while the magnitude of z is less than 2 or until an upper limit on the iterations is reached. The selected point is colored according to the number of iterations performed on the point. Points for which M did not diverge are considered members of the set. The color assigned to points that diverge indicates how far out of the set the point is.

Since each point can be tested independently of all others in the selected region, maximum parallelism would be achieved if each point could be simultaneously assigned to a separate processor. Optimal parallelism would require fewer processors, because some points require more time to compute than others. Hardware and software restrictions indicate what the granularity of parallism and work units will be.

The X Window System does not deal very readily with individual pixels of a window. Instead, X designates a rectangular collection of pixels, a *pixmap*, as a displayable resource[1]. This restriction together with the limitation on the number of available processors (< 16) circumscribes the number of pixels grouped together for a pixmap. With the window size known and a hint from the user regarding the number of work units, the selected region of the complex plane is partitioned into equal sized patches that will form the basic unit of work. Typically, the display window is 400×400 pixels partitioned into 40×40 pixmaps resulting in 100 work units (these are defaults).

The MSG program is invoked with these descriptive parameters and another that designates how many processors to use in the computation. The initially invoked process assembles the collection of processors to compute the set, presets the work unit allocation structures, creates the display window, and starts up client processes to perform the computation. The initial process will not do any set computation, simply manage the display window.

The display window (Figure 1) has two parts, the area for the computed pixmaps and a status line that indicates the elapsed time and percent of the picture that has been completed. The task of the initial process is to update this status line and refresh the display window as needed. As each available processor (worker process) becomes idle, it obtains the description of a patch that needs to be filled, performs the Mandelbrot set algorithm on each pixel of the patch, constructs the resulting pixmap, and sends the completed patch to the display node.

[1] A pixmap could be sized 1 by 1 pixel but pixmaps are a limited resource and a substantial picture would exhaust the available supply.

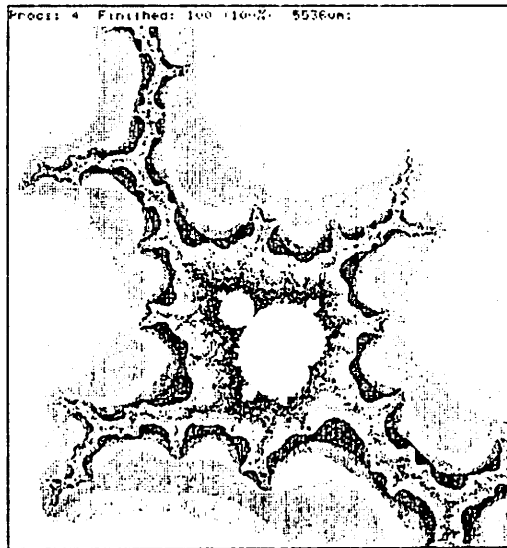

Figure 1: The Mandelbrot set display window

2.1 Excruciating Detail

The executing MSG program consists of the initial process that manages the display and the worker processes that compute the set. Each worker process regulates its own behavior as it acquires a patch, fills it, and displays (one time only) the result. The processes interact through shared data structures that describe completed patches and work yet to be done, and by their access to the X windows display connection. There is no other necessary process synchronization.

2.1.1 Shared Data Structures and Their Locks

The `RegionList` (figure 2) vector contains an entry for each work unit. The entry describes the completion status and the X Windows resource identifier for its resultant displayable pixmap. This structure is allocated and preset by the initial process when the number of patches is known. A worker process updates the proper patch entry when it acquires the patch and again when it completes the patch and has sent it to the display. The initial process scans this list whenever it services a window exposure event and redisplays any that are marked as done. It is not necessary to synchronize access to this vector; writers access only the entry they have been allocated.

The `FinishedPatches` variable is a count of the number of valid entries in the `RegionList`. Worker processes increment this counter when they add a pixmap to the `RegionList`. The initial process reads `FinishedPatches` to determine the percentage of complete patches. This variable

```
enum region_status {yettodo, inprogress, done};

shared int NextToDo =0;         /* next patch to dole out */
shared struct region_desc {
    enum region_status status; /* status of the region */
    Pixmap patch_id;           /* the values of the pixels in the region */
    } *RegionList;

shared int FinishedPatches;    /* incremented for each pixmap added to RegionList */

/*
 * locks to sync access to the display, and the task allocation model
 */
shared slock_t Xlock;           /* access to Xwindows */
shared slock_t queuelock;       /* the region list */
shared slock_t countlock;       /* the finished patches variable */
```

Figure 2: Important Shared Data

```
NextRegion ()

{   int i;

    /*
     *  lock the work unit allocator and obtain the next available patch
     */
    s_lock (&queuelock);
    i = NextToDo < Region_Count ? NextToDo++ : -1;
    s_unlock (&queuelock);
    if (i != -1) RegionList[i].status = inprogress;

    ebbaPostEvent (e_PatchAllocated, getpid(), i, 0, 0);
    return i;
}
```

Figure 3: Work unit allocation routine

is protected by the countlock mutual exclusion lock because it has potential for multiple writers. The countlock is acquired, FinishedPatches is incremented, and the lock released by the StorePatch() routine.

A work unit is allocated to a worker process when the worker calls the NextRegion() function (Figure 3). Work units are allocated in a left-to-right, bottom-to-top order by simply taking the current value of the NextToDo variable. Since each worker could execute NextRegion() concurrently

the `NextToDo` variable is protected by the `queuelock` mutual exclusion lock. The returned value determines which `RegionList` entry is relevant for the worker, implicitly providing the worker with exclusive write to the entry. When no more work units are available (`NextToDo` = total work units) `NextRegion()` returns a -1 value.

The connection to the remote X Windows server is shared by the initial process as well as the worker processes. The X interface library in use for this example does not support locking of the connection for mutual exclusion and so an explicit lock, `Xlock`, is provided for this purpose. This lock is acquired by a process each time an X Windows write request is made. Workers acquire the connection only in the `StorePatch()` procedure to create a pixmap from just completed patch data and to initially display the patch. The initial process acquires the connection in both the `UpdateStatusLine()` and `ServiceWindow()` routines.

2.1.2 Initial Process Task

The initial process task is responsible for presetting any global data structures, such as the variables describing the region to be investigated or the patch description list, and creating the worker tasks to perform the Mandelbrot Set Generator parallel algorithm.

There is nothing complicated about creating the worker tasks and coaxing them to compute parts of the Mandelbrot Set. The initial process simply forks a new process (figure 4) which then calls the `FillRegions()` routine. Upon return from `FillRegions()` each child process exits. Later, the initial process will collect all of the children and discard their exit status.

Following the setup phase, while the workers are computing the set of interest, the initial process task loops to update the status display and service any input X window events. `UpdateStatusLine()` formats the text for the status line, acquires `Xlock`, outputs the status text line to the status display, and releases `Xlock`. `ServiceWindow()` uses a non-blocking check on the X windows event queue (*XPending()*) to determine if there has been some mouse button activity (indicating that the user wants to select a region of the completed set) or if the display needs to be refreshed due to a window exposure. If some parts of the set display have become exposed, any completed pixmaps that cover that region are written to the display. The usual lock-write-unlock protocol is followed.

2.1.3 Worker task

Following creation each worker task executes the `FillRegions()` routine as its main process loop. The worker requests a work unit, then calls the work routine `FillPatch()` to test the points covered by the work unit patch and fill in the its private patch structure. `FillPatch()` is given an origin and the worker's patch structure. `FillPatch()` reads global parameter values that describe

```
Set X Window Server Connection;
Compute Geometry and Create the display Windows;
Preset Shared Structures;

/*
 *  create the processes that will do the stuff
 */
for (i = 0; i < N_proc; i++)
    if (fork() == 0) {
        FillRegions (i);
        exit (0);
        }
ebbaPostEvent (e_WorkersForked, N_proc, 0, 0);
.
.
.
/*
 *  service the display windows
 */
do {

    UpdateStatusLine ();
    if (XPending ()) ServiceWindow ();

    } while ((FinishedPatches != Region_Count) && Waiting_To_Quit);
.
.
.
/*
 *  collect the children
 */
while (wait (&status) != -1) ;
```

Figure 4: Initial Process Task

```
FillRegions (id)
  int id;

{   int index;
    double org_real, org_imag;
    u_char *patch;

    ebbaPostEvent (e_WorkerStart, getpid(), 0, 0);

    /*
     *  allocate some space for the patches to be filled.
     */
    patch = (u_char *)malloc
               (sizeof (*patch) * Pix_Width_Per_Patch * Pix_Width_Per_Patch);

    /*
     *  ask for an undone region or quit
     */
    while ((index = NextRegion()) != -1) {
        ebbaPostEvent (e_TakePatch, getpid(), index, 0, 0);
          .
          .
          .
        /*
         *  do the patch
         */
        FillPatch (org_real, org_imag, patch);
        StorePatch (index, patch);
        }

    ebbaPostEvent (e_WorkerDone, getpid(), 0, 0);

    free (patch);
    return;
 }
```

Figure 5: worker task main loop

the extent of the patch to execute the Mandelbrot Set generator algorithm

```
foreach point covered by the patch
        n = 0;
        z = 0; c = point to test;
        while n < max-iterations & magnitude(z) < 2
                z = z^2 + c;
                n = n + 1;
        patch[a][b] = colormap(n);
```

The `patch` vector owned by each process's `FillRegions()` routine is returned with the color values that characterize each point in the patch.

`StorePatch()` is invoked to change the raw color coded point values into a displayed pixmap. `StorePatch()` acquires the `Xlock`, creates a displayable pixmap from the raw patch data (by calling *XStorePixmapZ()*), and displays the pixmap (*XPixMapPut()*). The *XFlush()* routine is invoked to ensure that the pixmap is put on the display. The `Xlock` is released following the buffer flushing. `RegionList` is updated without any locking since only the entry corresponding to the completed patch is accessed, and no other worker process could be attempting to fill the same patch. The `FinishedPatches` variable is locked because contention by multiple writers could leave it in an incorrect state.

2.2 Instrumentation for Behavioral Abstraction

Instrumentation of the MSG program was initially undertaken to test the low-level event generation routines implemented for Symmetry/Dynix. The primitive events characterize the basic behavior of the program rather than give an exhaustive description of each step in its execution. This set of events would be typical if the user was interested in **What** the behavior of the MSG program was; rather than **How** that behavior is implemented. The program generates the following primitive events (represented by their templates):

(`e_WorkersForked` *count time location*)
: Created in the initial process following the creation all of the worker processes (figure 4). *count* is the total number of worker processes created.

(`e_WorkerStart` *id time location*)
: This happens in each worker following creation, and is the first notion that a new worker is up and running (figure 5). *id* is the worker process' process identifier.

(`e_WorkerDone` *id time location*)
: Created after a worker discovers that the work queue is empty and it should retire (figure 5).

(`e_PatchAllocated` *patchidx time location*)
: Issued by the patch allocation routine whenever it sends out a patch description; *patchidx* is

```
StorePatch (index, patch)
  int index;
  u_char *patch;

{   Pixmap pm;
    .
    .
    .
    ebbaPostEvent (e_DisplayPatch, getpid (), index, 0, 0);

    /*
     *  create the patch and force it to be output to the display
     */
    s_lock (&Xlock);
    pm = (Pixmap)XStorePixmapZ (Pix_Width_Per_Patch,Pix_Width_Per_Patch, patch);

    XPixmapPut (Mandel_Win,
                0, 0,
                (index % Regions_Per_Side) * Pix_Width_Per_Patch,
                Win_Extent - (index / Regions_Per_Side + 1)
                                         * Pix_Width_Per_Patch,
                Pix_Width_Per_Patch, Pix_Width_Per_Patch,
                pm, GXcopy, AllPlanes
                );
    XFlush ();
    s_unlock (&Xlock);

    /*
     *  update the global status information
     */
    RegionList[index].patch_id = pm;
    RegionList[index].status = done;

    s_lock (&countlock);
    FinishedPatches += 1;
    s_unlock (&countlock);
}
```

Figure 6: Store Patch routine

simply the index of the patch (Figure 3). The value is from 0 to $n-1$ where n is the number of patches to be filled.

(`e_TakePatch` *id patchidx time location*)
In `FillRegions()` a worker announces its intention to fill a patch. This is not symmetric with `e_PatchAllocated` since `NextRegion()` generates `-1` to signal that no work units are available.

(`e_DisplayPatch` *patchidx time location*)
Worker signals that a completed patch is being sent to the display. This is generated before the actual work is performed in getting the patch displayed.

2.3 All This Precision and Death Was Swift

The MSG program functioned in a satisfactory manner, achieving anticipated speedups (nearly linear) from the multiprocessor. Occasionally the program would fail unexpectedly, but these infrequent failures could be generally attributed to resource limitations. Using the instrumentation and observing behavior resulted in some minor changes to the instrumentation suite and some simple but demonstrative high-level behavior models.

However, it appeared that the VAXstation used to display the generated set suffered a severe performance degradation. Based solely on general knowledge about X Windows and the UNIX IPC that connects the MSG program to the VAXstation display, it was surmised that the rapid modification of the status line at the top of the display was responsible for the poor performance. A simple test for this hypothesis was to comment out the `UpdateStatusLine()` call of the initial task (figure 4).

And then the program broke.

Figure 7 is typical of the event stream received by the *EBBA* tools for a failing execution of the MSG program. This stream is not substantially different from streams observed in executions that proceeded to normal completion (except it is quite a bit shorter). With the simple change in place, the MSG program always failed with roughly the same behavior, but the details of the failure were not always the same—always at least one displayed patch, but never many.

Some simple observations can be made from this stream. All created workers did some useful work and attempted to display the results. One of the workers was allocated a second patch, which it completed and attempted to display. There are no patches that are allocated but not yet completed (the `e_DisplayPatch` event can be used as a guide that all the computation is completed on a patch).

```
Init: ...Eval, ...Library...Monitor, ...EvtQ, ...Interface, done!

EventMonitor startup at 18:19:32.07
Event Queue at 18:20:55.43

 1 (e_WorkerStart 8225 18:20:38.10 "min")
 2 (e_PatchAllocated 8225 0 18:20:38.10 "min")
 3 (e_TakePatch 8225 0 18:20:38.11 "min")
 4 (e_WorkerStart 8226 18:20:38.13 "min")
 5 (e_PatchAllocated 8226 1 18:20:38.13 "min")
 6 (e_TakePatch 8226 1 18:20:38.13 "min")
 7 (e_WorkerStart 8227 18:20:38.16 "min")
 8 (e_PatchAllocated 8227 2 18:20:38.16 "min")
 9 (e_TakePatch 8227 2 18:20:38.16 "min")
10 (e_WorkersForked 4 18:20:38.19 "min")
11 (e_WorkerStart 8228 18:20:38.19 "min")
12 (e_PatchAllocated 8228 3 18:20:38.19 "min")
13 (e_TakePatch 8228 3 18:20:38.19 "min")
14 (e_DisplayPatch 8225 0 18:20:38.29 "min")
15 (e_DisplayPatch 8226 1 18:20:38.34 "min")
16 (e_DisplayPatch 8227 2 18:20:38.43 "min")
17 (e_DisplayPatch 8228 3 18:20:38.48 "min")
18 (e_PatchAllocated 8225 4 18:20:39.14 "min")
19 (e_TakePatch 8225 4 18:20:39.14 "min")
20 (e_DisplayPatch 8225 4 18:20:39.45 "min")
```

Figure 7: Failing event stream for 4 workers

3. The First Pass - It's a Non-Terminating Critical Section

None of the processes have aborted. A reasonable first guess is that they are all waiting on some shared resource–that the initial process provides in the `UpdateStatusLine()` routine. In some way the absence of this call has introduced what appears to be a classic eager workers, but non-terminating critical section–Livelock. The work allocation scheme, access to the X Windows display channel, and the work queue status indicators are several elements of the MSG program that involve critical sections to properly access shared resources.

Examining the trace of Figure 7 this type of error is entirely reasonable. Each worker has apparently allocated, filled, and attempted to display a patch of the selected region. One has even performed on a second patch; none of the others have made this progress however. The workstation display only shows a single patch output–but this could be due to buffering of the X Window System request messages.

This section explains the source of errors directly related to synchronization mechanisms, the work involved in instrumenting the parallel processing library to illuminate possible synchronization errors, and finally a set of high-level models that demonstrate to the debugging tool user that the error is found elsewhere.

3.1 Implementation and Use of Critical Sections

All critical regions in the MSG program are implemented using spin-locks (busy waiting) to coordinate access to the code of the critical region. A process that successfully acquires a spinlock gains exclusive accesses to a code segment until it realeases the lock. Work allocation (Figure 3) is guarded by the queuelock structure. Access to the X client-server connection for output of a completed patch (Figure 6) or a new status line is guarded by the `Xlock` spin-lock. The `countlock` protects the statement that bumps the finished patches counter.

The normal, correct behavior of a worker can be described by the model of Figure 8. After it starts, the worker cycles, obtaining, filling, and outputting patches until it is given a sentinel patch ($patchidx = -1$). All of the critical regions are used by a worker following the e_`DisplayPatch` and before the next e_`TakePatch`. It is reasonable to ignore the critical sections for the work queue and the finished patches counter–unless the locks are broken. Since the `DisplayPatch` event is output *before* the patch output critical region is entered it is possible that one of the combatants has somehow entered the section in `StorePatch()` and not returned. But WORKER-1[2] has proceeded through the critical section apparently unmolested, why would no other worker be allowed to do

[2]Who the workers are is easily determined by examining the *pid* fields of events in the stream. The lowest is the initial task; workers 1 through n get increasing process identifiers following the initial task. Similarly, which lock–given by address–corresponds to the program defined entities is simple. More on all this later.

```
event CorrectBehavior is
    e_WorkerStart ∘ (e_TakePatch[1] ∘ e_DisplayPatch)*∘ e_TakePatch[2] ∘
  e_WorkerDone
cond
  e_WorkerStart.id == e_TakePatch[1].id;
  e_WorkerStart.id == e_DisplayPatch.id;
  e_WorkerStart.id == e_TakePatch[2].id;
  e_WorkerStart.id == e_WorkerDone.id;

  e_TakePatch[1].patchidx == e_DisplayPatch.patchidx;
  e_TakePatch[2].patchidx == -1
with
  Worker: integer :=e_WorkerStart.id
end
```

Figure 8: Model for correct high-level Patch filling

the same (maybe WORKER-1 damaged the lock)?

The behavioral possiblities that bear investigation now include:

- Did the first worker through damage the locks so that no process is subsequently able to acquire a lock and all processes are waiting to enter? Likewise, have two processes gained access and destroyed some other piece of information?
- In the mold of classic synchronization and Livelock errors, has a worker entered a critical region and not left; causing all other workers to wait at the entrance?

To investigate the worker interaction at the critical regions it is sufficient to change viewpoints and examine the system in terms of the spin locks used to implement the critical regions.

3.2 Instrumentation of the Mutual Exclusion Lock Routines

In the Sequent Symmetry series spin-locks are implemented using simple memory locations. Correct operation of the locking algorithm is guaranteed by the hardware cache/bus/memory protocol. Convenient access to spinlocks is implemented by the Sequent Parallel programming library through three operations:.

`s_init_lock`(*lock-var*) puts the spin lock given by *lock-var* into a known state. Failure to call *s_init_lock()* before using a lock variable will have unpredictable results.

`s_lock`(*lock-var*) attempts to acquire the lock given by *lock-var*. If the call does not acquire the lock, the calling process busy waits until it is successful.

```
s_lock(lock)
  register slock_t *lock;
{
  S_LOCK(lock);
#ifdef EBBA
  ebbaPostEvent (e_LockAcquired, getpid(), lock, 0);
#endif EBBA
}
```

Figure 9: Instrumented Library routine

`s_unlock`(*lock-var*) frees the lock for another process to acquire.

Corresponding to these three operations are three primitive events generated as these routines execute:

(`e_LockPreset` *id lockname time location*)
: Created when a process calls *s_init_lock()* to set a spinlock to its known unlocked value. *id* is the process identifier for the process setting the lock; *lockname* is the virtual address of the lock structure in shared memory. *lockname* is a unique name for a lock properly configured among a group of cooperating processes since it is mapped into the same shared memory.

(`e_LockAcquired` *id lockname conditional time location*)
: This event is generated when a process calls *s_lock()* to acquire a spinlock and has successfully completed the call. The *conditional* attribute is 1 if the lock request was conditional (called via *s_clock()*). *id* and *lockname* have the same meaning as in `e_LockPreset`.

(`e_LockReleased` *id lockname time location*)
: The event generated during calls to *s_unlock()*. This event is created before the actual lock release, in order to avoid confusing acquired/released misorderings.

The annotations required to generate these primitive events were added to the source text of the parallel programming library. The code fragment of Figure 9 is typical of the instrumentation annotations added to the parallel programming library. `S_LOCK` is a macro that expands into a machine-language sequence that performs the actual variable access and busy waiting. The library was subsequently recompiled and used for instrumented runs of parallel programs.

3.3 Simple Investigation of Critical Section Behavior

The behaviors outlined in section 3.1 could be investigated in terms of the lock instrumentation. A simple high-level model, `LockPair` (Figure 10), can be used to observe the behavior and determine:

```
event LockPair is
      e_LockAcquired o e_LockReleased
cond
      e_LockAcquired.lockname == e_LockReleased.lockname;
      e_LockAcquired.id == e_LockReleased.id
end
```

Figure 10: LockPair high-level event

1. If there is an `e_LockAcquired` that has no corresponding `e_LockReleased`, then a worker has entered a critical region and not left, If no worker is able to acquire a lock then there will be no outstanding `e_LockAcquired` events.
2. the locks are not functioning correctly and there are two workers inside a critical region if there are two `e_LockAcquired` events without an intervening `e_LockReleased` on the same lock.

A request to monitor the stream for all the occurrences of this model recognizes a collection of `LockPair` events and a single incomplete `LockPair` event[3]. The listing of Figure 11 contains some detail for each recognized `LockPair` event and the partial interpretation for the incomplete `LockPair` (not all partial interpretations are shown). Completed events are listed in most-recent-first order displayed with the events bound to the model as constituent events. The listings of completed model interpretations include the representation of the instance that resulted from the recognition and the event set (model constituents) that was bound to the model when it was recognized. Partial interpretations indicate the state of the recognizer and the set of events that are bound to the interpretation at this point.

The completed interpretations demonstrate that the locks surrounding the critical sections are properly nested and paired. Also, the locking routines appear to be doing their job. The livelock explanation remains feasible but has no obvious cause. The single partial interpretation of Figure 11 indicates that the `Xlock` guarding the `StorePatch()` critical section has been acquired but not released. It appears that a worker (WORKER-2) has entered and has not exited, while the other workers remain viable. Examination of that routine (Figure 6) shows that there are no loops at the application level that the worker process might be cycling endlessly. This critical region needs to be more closely examined since, the evidence is that one of the X Window System interface library routines is at fault, in some way.

[3] Which lock is given by the *lockname* field of each event; the correspondence to program names is: `...f4-Xlock`, `...f6-queuelock`, `...f8-countlock`

```
Status for LockPair, lp-1

queue position 0x43304
LockPair <16384>
interest set   e_LockAcquired e_LockReleased.
input set coding [ 8 7 ]
Partial Instantiations: 8
Completed Recognitions: 8

Completed :
(LockPair 8340 0x230f6 20:39:00.67 "EventMonitor")
Constituents
    0) (e_LockAcquired 8340 0x230f6 0 20:39:16.05 "min")
    1) (e_LockReleased 8340 0x230f6 20:39:16.05 "min")

(LockPair 8340 0x230f8 20:39:00.09 "EventMonitor")
Constituents
    0) (e_LockAcquired 8340 0x230f8 0 20:39:16.05 "min")
    1) (e_LockReleased 8340 0x230f8 20:39:16.05 "min")

(LockPair 8340 0x230f4 20:38:59.60 "EventMonitor")
Constituents
    0) (e_LockAcquired 8340 0x230f4 0 20:39:15.90 "min")
    1) (e_LockReleased 8340 0x230f4 20:39:16.05 "min")

(LockPair 8343 0x230f6 20:38:59.03 "EventMonitor")
Constituents
    0) (e_LockAcquired 8343 0x230f6 0 2020:39:14. "min")
    1) (e_LockReleased 8343 0x230f6 20:39:14.99 "min")

(LockPair 8342 0x230f6 20:38:58.66 "EventMonitor")
Constituents
    0) (e_LockAcquired 8342 0x230f6 0 20:39:14.96 "min")
    1) (e_LockReleased 8342 0x230f6 20:39:14.96 "min")

(LockPair 8341 0x230f6 20:38:56.86 "EventMonitor")
Constituents
    0) (e_LockAcquired 8341 0x230f6 0 20:39:14.93 "min")
    1) (e_LockReleased 8341 0x230f6 20:39:14.93 "min")

(LockPair 8340 0x230f6 20:38:56.53 "EventMonitor")
Constituents
    0) (e_LockAcquired 8340 0x230f6 0 20:39:14.90 "min")
    1) (e_LockReleased 8340 0x230f6 20:39:14.90 "min")
 => "lp-1"

Partial interpretations:
State 2
Event set instances:
  0) (e_LockAcquired 8341 0x230f4 0 20:39:16.05 "min")
  1) -- undef --
```

Figure 11: Summary of LockPair investigation

4. X Window Interface as the Defendant

It is necessary to more closely investigate the critical section in the `StorePatch()` routine. This critical section consists entirely of calls to X Window System interface library routines. It is unlikely that the fault is to be found at the application layer of X, but to determine if a specific routine is causing difficulty and to provide a high-level hook to hang lower level behavioral models from, it is necessary to examine the X interface library from the application level.

Investigation in terms of the X windows viewpoint also makes it possible to observe interaction of the initial task with its workers.

4.1 X Windows Instrumentation

The X Window System interface library was annotated with `ebbaPostEvent()` calls to generate the following (among other) events:

(`e_XFlush` *pid time location*)
: When the user calls *XFlush()*, an internal flush, *_XFlush()* is invoked to write any buffered requests. The event is generated following return from the internal call,

(`e_XNextEvent` *type window_id pid time location*)
: Created when *XNextEvent()* returns a queued event to the client,

(`e_XOpenDisplay` *displayname pid time location*)
: Created after the client successfully opens the connection to the server,

(`e_XPending` *qlen pid time location*)
: Created following a read on the server connection, (but not the case where a read is not performed),

(`e_XPixmapPut` *seqno window_id pixmap_id pid time location*)
: Created following the buffering of the `X_PixmapPut` server request,

(`e_XSelectInput` *seqno window_id mask pid time location*)
: Created following the buffering of the `X_SelectInput` server request,

(`e_XText` *seqno pid time location*)
: Created following the buffering of the `X_Text` server request and its associated data,

(`e_XStorePixmapZ` *seqno pixmap_id pid time location*)
: Created following the exchange that obtains the resource id for the new pixmap.

This event library is merged into the viewpoint through which the MSG program is currently being observed, giving a new viewpoint in terms of the original Mandel primitive events, the locking mechanism primitive events, and now the X interface library primitive events. Merging is easily performed by the Librarian component of the toolset. The user buttons a "merge library" menu item

```
event PatchStored is
        e_PatchAllocated ∘ e_DisplayPatch ∘ e_LockAcquired ∘ e_XStorePixmapZ ∘
    e_XPixmapPut ∘ e_XFlush ∘ e_LockReleased
cond
    e_DisplayPatch.id ==e_PatchAllocated.id;
    e_LockAcquired.id ==e_PatchAllocated.id;
    e_XStorePixmapZ.pid ==e_PatchAllocated.id;
    e_XPixmapPut.pid ==e_PatchAllocated.id;
    e_XFlush.pid ==e_PatchAllocated.id;

    e_LockReleased.id ==e_PatchAllocated.id;
    e_DisplayPatch.patchidx ==e_PatchAllocated.patchidx;

    e_LockAcquired.lockname ==e_LockReleased.lockname
with
    patchidx: integer :=e_PatchAllocated.patchidx;
    pid: integer :=e_PatchAllocated.id
end
```

Figure 12: `PatchStored` high-level model

and supplies the name of a library to merge. The librarian interface notifies interested components of the change and they update their viewpoints correspondingly.

4.2 Looking Through the X Windows Viewpoint

A more complete model for the behavior necessary for creating and displaying a patch is seen in Figure 12. Examining the event stream through this model produced the results of figure 13. These confirm earlier results and correlate with the observed output to the VAXstation display. One patch was completed (`PatchStored`.*patchidx* = 0) and a single worker is in its critical section while the others await their turn. The important part to note about the critical section holder (the fourth partial interpretation) is that there are no X Windows events in its event set. The implementation of (at least) the X Interface Library routine *XPixmapStoreZ()* is responsible for the lack of progress, in some way.

With the view through X windows it is now possible to observe behavior in the initial task beyond its initial lock presets and attempt to explain how interaction with the workers causes the program to fail. The initial task behavior (without the `UpdateStatusLine()` call) can be partly described by the event description

```
Completed Interpretations, PatchStored, Ps-1
(PatchStored 0 12725 15:52:32.79 "EventMonitor")
Constituents
    0) (e_PatchAllocated 12725 0 15:48:45.62 "min")
    1) (e_DisplayPatch 12725 0 15:48:46.67 "min")
    2) (e_LockAcquired 12725 0x290f4 0 15:48:46.67 "min")
    3) (e_XStorePixmapZ 80 0xbf4035a 12725 15:48:46.89 "min")
    4) (e_XPixmapPut 81 0xbf30389 0xbf4035a 12725 15:48:46.90 "min")
    5) (e_XFlush 12725 15:48:46.90 "min")
    6) (e_LockReleased 12725 0x290f4 15:48:46.90 "min")

Partial Interpretations, PatchStored, Ps-1
State 3
Event set instances:
  0) (e_PatchAllocated 12725 4 15:48:46.91 "min")
  1) (e_DisplayPatch 12725 4 15:48:48.08 "min")
  2) -- undef --
  3) -- undef --
  4) -- undef --
  5) -- undef --
  6) -- undef --

State 3
Event set instances:
  0) (e_PatchAllocated 12728 3 15:48:45.77 "min")
  1) (e_DisplayPatch 12728 3 15:48:46.86 "min")
  2) -- undef --
  3) -- undef --
  4) -- undef --
  5) -- undef --
  6) -- undef --

State 3
Event set instances:
  0) (e_PatchAllocated 12727 2 15:48:45.71 "min")
  1) (e_DisplayPatch 12727 2 15:48:46.81 "min")
  2) -- undef --
  3) -- undef --
  4) -- undef --
  5) -- undef --
  6) -- undef --

State 4
Event set instances:
  0) (e_PatchAllocated 12726 1 15:48:45.67 "min")
  1) (e_DisplayPatch 12726 1 15:48:46.79 "min")
  2) (e_LockAcquired 12726 0x290f4 0 15:48:46.91 "min")
  3) -- undef --
  4) -- undef --
  5) -- undef --
  6) -- undef --

State 3
Event set instances:
  0) (e_PatchAllocated 12725 0 15:48:45.62 "min")
  1) (e_DisplayPatch 12725 0 15:48:46.67 "min")
  2) -- undef --
  3) -- undef --
  4) -- undef --
  5) -- undef --
  6) -- undef --
```

Figure 13: Results of `PatchStored` high-level model

```
event initial is
        e_XOpenDisplay o (e_XPending o e_XNextEvent)+
cond
     e_XPending.qlen !=0;
     e_XNextEvent.type ==ButtonPressed || e_XNextEvent.type ==ExposeWindow
end
```

This model shows that the INITIAL process opens the client-server connection, then executes *XPending()/XNextEvent()* pairs whenever there is an input X Windows event for the client to process. Examining the event stream in terms of this model with the constraints (cond relational clauses) relaxed it can be seen that the initial process behavior is not as expected–it creates an e_NextEvent with its *type* attribute equal to NoEvent (value, 0):

```
Completed Interpretations, initial, initial
   (initial 15125 22:02:25.39 "EventMonitor")
   Constituents
      0) (e_XOpenDisplay "grinch:0" 15125 22:00:19.20 "min")
      1) (e_XPending 1 15125 22:00:20.68 "min")
      2) (e_XNextEvent 0 0xd8b036d 15125 22:00:20.68 "min")
```

The NoEvent X Windows event type is a reply to a resource creation request. Apparently the initial task has intercepted the reply packet destined for one of the worker processes. This behavioral observation is key evidence to the error.

There now may be sufficient information to correct the program. Ignorance about how input events (from the server) could interact with the request/reply requirements of resource creating library functions led to a programming decision to not lock the X server connection for servicing events. A demonstration of initial process-worker process interaction that involves this key behavior will solidify a proposed solution, without simply treating the error symptoms.

Initial-worker process interaction is at the implementation level of the X Interface Library. *XStorePixmapZ()* is outlined in Figure 14. The GetReq() and Data() macros load the client-server connection buffer with the protocol request (X_StorePixmap) and its data (the pixels that make up the pixmap). The internal (to the library interface) routine, *_XReply()*, will force the buffer to be output and call *_XRead()* to accept the pixmap resource id returned by the server. *_XRead()* contains a blocking read on the client/server connection. If the initial task has stolen the reply then a worker executing *XStorePixmapZ()* will block awaiting another reply. But another reply is not possible since all the other worker tasks are waiting at the XLock to send their most recently completed patch to the server.

XPending() is outlined in Figure 14. It first ensures that any buffered X requests are sent (*_XFlush()*), then tests the channel to determine if any data is present using *ioctl(2)*. The *ioctl()* will return a value (pend) if the server has sent events (including resource creation replys) to the

```
Pixmap XStorePixmapZ (width, height, data)
    int width, height;
    caddr_t data;
{
    GetReq(X_StorePixmap, 0);
      . . .
    Data (dpy, data, len);
    if (!_XReply(dpy, &rep)
      . . .

    return (rep.param.l[0]);
}
```

```
int XPending ()
{
      . . .
    _XFlush (dpy);
    if (ioctl(dpy->fd, FIONREAD, &pend) < 0)
      . . .

    _XRead (dpy, buf, pend);
      . . .

    return(dpy->qlen);
}
```

Figure 14: *Outlines for XStorePixmapZ()* **and** *XPending()*

MSG program. A non-zero pend causes the Initial process to enter the blocking read of *_XRead()*. Conditions are quite ripe for a race to the blocking read.

To observe this interaction, the internal *_XRead()* function is instrumented in terms of the generic procedure entry/exit viewpoint[4]. The `D_Shuffle` model of figure 16 captures the interaction of the initial process and its workers at this point. Observed through this model two completed abstractions result (Figure 15). Each is a valid interpretation but only one is relevant.

Both of these interpretations are valid because the interaction of three processes (INITIAL, WORKER-1, and WORKER-3 in this case) is concurrent and overlap in time. The system can be viewed through several other models to help sort this out. The completed `PatchStored` model provides evidence that the `D_Shuffle` with $pid = 3516$ and $patchidx = 0$ does not represent an interaction gone awry. Further evidence is provided by the `ProperIlv` (proper interleaving) model that demonstrates that the entry and exit from the read to retrieve the resource id are matched for WORKER-1. A partially matched `ProperIlv`:

```
State 3
   Event set instances:
     0) (e_DisplayPatch 3518 2 17:10:51.59 "min")
     1) (e_procEntry "_XRead" 3518 17:10:51.63 "min")
     2) -- undef --
```

provides direct evidence that WORKER-3 began its read but did not complete.

This apparent ambiguity is caused in part because there is no way to express direct causality from the available events. Causality can only be directly shown for events created by a single process. Causality among interacting concurrent processes is, at best, difficult owing to the lack of a single causal authority (the program counter suffices for one processor systems without inter-

[4]This instrumentation can be added at compile time by a modified version of the Sequent C compiler.

```
Completed Pending Events

OKiIv-1, HighLevel
(ProperIlv 3516 0 17:07:41.84 "EventMonitor")
Constituents
    0) (e_DisplayPatch 3516 0 17:10:51.47 "min")
    1) (e_procEntry "_XRead" 3516 17:10:51.48 "min")
    2) (e_procExit "_XRead" 3516 17:10:51.61 "min")

D_S-1, HighLevel
(D_Shuffle 3516 0 17:07:40.74 "EventMonitor")
Constituents
    0) (e_XOpenDisplay "grinch:0" 3515 17:10:50.14 "min")
    1) (e_DisplayPatch 3516 0 17:10:51.47 "min")
    2) (e_procEntry "_XRead" 3515 17:10:50.14 "min")
    3) (e_procEntry "_XRead" 3516 17:10:51.48 "min")
    4) (e_procExit "_XRead" 3515 17:10:51.96 "min")
    5) (e_XPending 1 3515 17:10:51.96 "min")
    6) (e_XNextEvent 0 0x1fc60395 3515 17:10:51.96 "min")

D_S-1, HighLevel
(D_Shuffle 3518 2 17:07:40.59 "EventMonitor")
Constituents
    0) (e_XOpenDisplay "grinch:0" 3515 17:10:50.14 "min")
    1) (e_DisplayPatch 3518 2 17:10:51.59 "min")
    2) (e_procEntry "_XRead" 3515 17:10:50.14 "min")
    3) (e_procEntry "_XRead" 3518 17:10:51.63 "min")
    4) (e_procExit "_XRead" 3515 17:10:51.96 "min")
    5) (e_XPending 1 3515 17:10:51.96 "min")
    6) (e_XNextEvent 0 0x1fc60395 3515 17:10:51.96 "min")

PS-1, HighLevel
(PatchStored 0 3516 17:07:38.11 "EventMonitor")
Constituents
    0) (e_PatchAllocated 3516 0 17:10:50.45 "min")
    1) (e_DisplayPatch 3516 0 17:10:51.47 "min")
    2) (e_LockAcquired 3516 0x250f4 0 17:10:51.47 "min")
    3) (e_XStorePixmapZ 80 0x1fc503a9 3516 17:10:51.61 "min")
    4) (e_XPixmapPut 81 0x1fc40237 0x1fc503a9 3516 17:10:51.61 "min")
    5) (e_XFlush 3516 17:10:51.62 "min")
    6) (e_LockReleased 3516 0x250f4 17:10:51.62 "min")
```

Figure 15: Telltale high-level event instances

```
event D_Shuffle is
        e_XOpenDisplay ∘ (e_DisplayPatch △ e_procEntry[1]) ∘ e_procEntry[2] ∘ e_procExit ∘
      e_XPending ∘ e_XNextEvent
cond
      e_DisplayPatch.id ==e_procEntry[2].pid;

      e_procEntry[1].pid ==e_XOpenDisplay.pid;
      e_procExit.pid ==e_XOpenDisplay.pid;
      e_XPending.pid ==e_XOpenDisplay.pid;

      e_XPending.qlen !=0;
      e_XNextEvent.type ==NoEvent
with
      pid: integer :=e_DisplayPatch.id;
      patch_index: integer:=e_DisplayPatch.patchidx
end
```

Figure 16: ***D_Shuffle*** **model for initial task/worker interaction**

rupts). A weakness in the high-level event description is also in part responsible for the interpretive ambiguity. There is no mechanism for expressing the absence (match only if something does *not* occur) or negating the presence (do not match if something *does* occur) of an event.

4.3 The Fix

This error can be explained at different abstraction levels, corresponding to the implementation layers of the MSG program. Each level provides a more detailed, accurate, explanation. From the perspective of the MSG program, the highest level, the error is that the initial task, executing its *XPending(); XNextEvent()* behavior, is mistakenly reading the reply to a worker's resource creation request. The initial task is not observing the locking protocol for access to the client-server connection. To fix the program, the *XPending(); XNextEvent()* sequence is protected by the `Xlock` variable. This will prevent the initial process from accessing the channel while a worker is awaiting a reply to its create pixmap request. This treats the symptoms. It does not explain the complex interaction that causes the error.

A better explanation is derived from the evidence presented in the previous section. The initial process executes its *XPending()* as the server reply to the worker *XStorePixmapZ()* request arrives. Noting that an event (the reply) is available, the initial process *_XRead()*'s this event. The initial process unceremonialy discards the event because it is not in the list of events it will service. The worker expecting a reply blocks while holding `Xlock`–leaving all its co-workers spinning and the initial process looping on *XPending()* calls. This explanation illustrates that the *_XRead()*(*read(2)*)

is really a destructive operation by virtue of its changing the status of the connection buffer.

Understanding the error in these behavioral terms has several benefits:

- the correction is not simply treating the error symptoms,
- the application writer can avoid similar reader interaction errors,
- understanding this behavior provides added value because it indicates how the X Interface Library might be changed to provide the required synchronized access to the client-server connection. Thus relieving the application from this duty and simplifying programming.

The previous explanation is still lacking since it appears that there is a very, very narrow time window that the initial process has in which it can steal the reply. A more detailed explanation, that completely covers the error involves the Dynix kernel as well as the MSG participants. The kernel maintains the client-server connection. As it receives data, the kernel buffers the input on the client-server socket and sets appropriate status in the socket descriptor. Each worker sends its pixmap create request then blocks via a *read(2)* on the socket. The initial process issues an *ioctl(2)* to determine if there are any events to read. This *ioctl()* returns true from the time the kernel buffers the input data until the worker is rescheduled, selected to run, and updates the pointers to indicate there is no longer data (completes the *read()*).

Close examination of the event stream shows that the initial process enters *_XRead()* while the first worker is performing its *XStorePixmapZ()* function. The *_XRead()* from the initial process occurs after that worker,so WORKER-1 completes its *XStorePixmapZ()* normally. However, the initial process is now blocked on the socket read request. The next worker that attempts to create a pixmap will block *behind* the initial process. The initial process steals the reply event from the worker and leaves the worker holding the `Xlock`.

5. Retrospective on the Instrumentation

Instrumentation is a critical issue for *EBBA*, as it is for many other analysis and debugging techniques that are under study. The purpose of instrumentation is to obtain the raw information that characterizes the system under study, in the detail and form required by the operation of the investigation tools. A desirable property of any instrumentation is that it be unobtrusive, i.e. it does not affect the instrumented software in any way. Completely unobtrusive instrumentation requires hardware and software that is totally execution independent of the monitored system.

Within the *EBBA* framework, instrumentation is responsible for generating the primitive event instances that are the basis for observing system behavior. In the absence of an independent hardware solution, this requires some *active* system object to note when the important transitions that constitute events have occurred, and create the record that marks this passage. Code segments

that consume storage space and cpu cycles must be added to the system under study to implement the active instrumentation.

The goals of the instrumentation used for *EBBA* have been to minimize the memory requirements for annotation code, minimize the execution time of each annotation, and support an environment that promotes flexible binding of probes to the monitored software. The first two items have been largely successful and experience has demonstrated the constraints and advantages of various approaches. The third goal has been achieved to only a limited extent. Mechanisms to aid instrumentation-software binding have been added to the Model builder toolset component, and the Sequent C language compiler has been modified to create a restricted class of primitive events. However, techniques for very flexible binding or simple probe activation/deactivation have yet to be achieved.

EBBA toolset support for event instrumentation involves the Model Builder component and an event formatting/delivery library (*libEBBA*). The current incarnation of the behavioral abstraction tools requires annotations (probes) added to source program code at points where execution constitutes behavior. Largely using library routines, the annotations marshall attributes, package instance records, and send these to the event monitoring tools. The role of the Model Builder is to create for each primitive event a descriptor that describes the shape and content of the primitive event. These descriptors connect annotations to primitive event instances.

There are many aspects to the general problem of instrumentation for any kind of behavioral visiblity. In the work on *EBBA* so far, there are several important instrumentation issues that have been considered:

(placement) policy determines where the probes are inserted into the executable program, what they illuminate, and what they are allowed to access as information,

probe effects will determine how the annotations alter the original program, both statically, as measured by additional program volume attributed to annotation code, and at runtime in terms of consumed process resources, particularily time (cpu cycles) and memory,

mechanism is how the probes actually work–which determines in part how they alter the program. An important part of the probe mechanism is how they are inserted into the software to be monitored. This has a large impact on how easy it is to instrument a program. Closely related is the technique employed for probe specification–how to describe what information the probe returns. Specification should also include placement information.

The remainder of this section will examine these issues in greater depth from the *EBBA* perspective.

5.1 Probe Policy

What to Choose as Primitive Events.

Adding instrumentation to a software component for *EBBA* is conceptually simple: identify the

important transitions of a component then annotate the component to illuminate these behaviors. Identification is generally easy since it follows directly from the functionality provided by a component. For example, the three Parallel Programming library routines that implement access to spinlocks are easily characterized by three primitive behaviors (Section 3.2). Similarly, the primitive events associated with the X Windows interface library characterize the behavior resulting from client interaction with the interface (Section 4.1). All of these primitive events are reasonably honest in their characterization of component behavior. But what the primitive events characterize for each component is quite different. The parallel programming library behaviors represent transitions in the state of a single variable. For the X interface library, each primitive event represents a summary of, in some cases considerable, lower level activity.

It would be difficult (but possible) to characterize the X Windows interface library behavior in terms of its implementation activity. This is due mainly to the extremely low behavioral level that consists largely of structure editing and an occasional, seemingly unconnected, system request. At this level of detail it is probably only cumulative effects that contain enough information to be useful. This is a clear example of when to instrument a component with its own set of primitive events versus when description of these same behaviors as high-level events is not reasonable.

On the other hand a behavioral view of the X windows protocol implementation could be easily characterized. The actual program activity that creates the protocol is equally low-level as the interface library. However, the clearly delineated activities associated with the protocol behavior would allow a "protocol implementation" event library to describe this protocol as well as others.

As antecdotal evidence it should be noted that all of the instrumentation that has been added to system software components is fundamentally on target. No events have required other, unanticipated, events to be defined to complete their behavior characterization. In a couple of cases, attributes required adjusting, but none was in response to a need for specialized information resulting from the accuracy of hindsight.

When is the behavior a primitive event?
The simple act of calling a routine does not imply that any significant behavior has taken place. For example, the `e_LockAcquired` event is created only when the calling routine successfully executes the code that permits the process to pass through the lock. A conditional lock request that is unsuccessful returns to its caller as unsuccessful, but no event is generated to characterize the attempt. The importance of placement is also seen in the `e_LockReleased` event. This event is created *before* the program activity takes place. Placing the annotation this way is an attempt to eliminate the appearance that released/acquired pairs are out of order.

A behavior might complete at more than one point in the execution of its defining component. This requires the annotation for a single event class to be inserted at a number of points in the program or possibley altering the original program in certain ways. For example the *XPending()*

routine generates events

(e_XPending *glen pid time location*).

The routine has several exits, one where there is no change in the status of the X Windows event queue, others for various detected error conditions, and another where X Windows events are available to be dequeued by the application. The (e_XPending ...) events are created only under the last of these conditions. The intent is to permit easy characterization of the major use of *XPending()* in X Windows programs:

```
while (XPending()) {
   XNextEvent (&e);
   switch (e.type) ...
```

Behavior modellers interested only in whether *XPending()* has been invoked are really interested in another behavioral viewpoint. They should use the program-level behaviors of the EntXit event library to provide this visibility.

Similar to this are events that result from an accumulation of very elemental activity. The primitive event should be created when a "key" piece of information has been created. A very situation specific judgement is sometimes required to do this correctly.

The carefully chosen wording above, "...characterize the behavior resulting from client *interaction* ...", underscores the need for a behavioral abstraction user to sometimes be aware of the precise semantics of the representative primitive events. Buffering effects or other sources of latency in the exchange between cooperating system components are often responsible for "virtual" behaviors. The aforementioned cumulative effects should be included here as well. For the X Windows example, request buffering introduces undetermined latency between request and actual server activity. Incorrect requests are not indicated to the client program until a round trip message exchange with the server is performed.

The difficulties that raise the issues for probe placement policy are not unique to the behavior abstraction approach. All debugging techniques require probes, and thus demand a probe policy. Behavior abstraction helps this process because its systematic approach guides probe insertion and relies heavily on reusability. Reusability comes in two forms for *EBBA*. Once an event annotation has been added to a software component, this effort is finished. Its behavioral characterization will not change so when it is needed a modeler can expect the behavior to be available. High-level models developed for a particular investigation can be reused at another time–e.g. the LockPair event.

5.2 Probe Effects

Time.
Instrumentation annotations necessarily alter code size and execution time characteristics of the program under study. Overall execution time increases. The amount of added execution time is directly related to the number of primitive events generated during the execution of a program. A large slowdown could be due to a small number of probes executing a large number of times; or a large set of probes, each executed a moderate number of times. Generating a single primitive event instance will add 2-3ms to the execution time of a program[5].

One of the advantages of *EBBA* is that the instrumentation can be varied according to the changing needs of the modeller. It is also sometimes possible to mitigate probe effects by shifting viewpoints. As an example, instead of suffering probe effects in a client due to X interface library primitive event generation, a suitably instrumented X Windows server could supply events to illuminate the same behavior; albeit from a slightly different viewpoint and with different effects on overall system behavior.

For debugging purposes, increases in overall execution time are seldom important. After all, the program is broken, having it run a bit longer is not all that serious. This general degradation in execution time is more important if the event probes are to be used to characterize program performance.

If increased execution time is a factor it is more likely because timing relations among program code sequences are altered due to the insertion of the time blocks that result from annotation execution. There are two kinds of time relations, each affected in different ways. Absolute time, characteristic of real-time systems, only admits carefully defined fixed time periods between system transitions. These time periods are often hard deadlines. Event generation annotations could clearly disrupt these relations both for individual time intervals and cumulatively as annotation executions cause the system to slip slightly. Cycle slippage in this type of system would certainly mask existing errors or introduce new errors.

Relative time relations, characteristic of programs that share information using signal and wait techniques, can be disrupted in more subtle ways. The major effect of annotation execution is to slightly change the execution ordering of interacting processes. For programs that enforce some rigid ordering regime, such as barrier synchronization, this is probably not a serious problem. However, for asynchronously interacting programs, such as the MSG program, this subtle reordering can mask the faulty behavior.

Memory Requirements.

---

[5]These times were obtained on the Intel i386 processors of the Sequent Symmetry systems by reading the hardware microsecond clock.

The executable program size increases by the volume of the code that implements the event generation. For a baseline, the annotations that generate the primitive events reported here require that two lines be added to the source code: the event generating `ebbaPostEvent(`*descriptor* ...`)` call and the line containing the primitive event *descriptor* text. The executable text of the `ebbaPostEvent()` call results in argument list evaluation and a procedure call. The static data area for the system under study grows slightly because of buffers necessary for instance formatting and the storage resulting from the primitive event *descriptor* strings.

Additional resources are required for the code that supports the annotations. Backing the current annotation scheme is approximately 12K bytes of library routine code added to the executable program image. This represents about 16% of the total executable image for the MSG program. The library size is constant, all programs annotated for the current event reporting scheme incur the same overhead. This could be reduced substantially (to around 5 kilobytes) if the event instance records were not encoded as text strings.

There are a number of causes of incidental overhead that probably cannot be evaluated in a meaningful way. Contention for I/O channels (for the event stream) and signalling facilities (for intervention and urgent notification effects) are affected. At runtime, program stack growth requirements increase due to executing event annotation code. These effects possibly will increase incidental overhead in unknown ways due to memory paging, processor cache behaviors, and system I/O requirements forced by executing the annotations.

Probe effects are an unavoidable consequence of the need to monitor a system in its own terms. The ONLY way to avoid probe effects is to use completely execution independent hardware, a solution that is expensive, does not readily allow reuse, generally only yields very low-level information, and can be equally overwhelmed by system execution. To mitigate the probe effect the best that can be done is to diligently minimize the interference caused by probes (this is wimpy, but all there is). This is effected by creating investigation tools that expect a minimal amount of information from probes with a regular structure. Regular structure minimizes the machinery needed to obtain the information. Minimal information requires minimal time to acquire. In an ideal environment, the investigation tools should be capable of dealing effectively with the subtle changes in system behavior caused by the probe effect.

5.3 Mechanism

What is required for an annotation?

Currently, primitive event instances are tied directly to the execution flow of a program. Events are generated on the basis of a component attaining a particular program counter value[6]. Mini-

[6] A more general "data relational calculus" that relates changes in multiple program entities is under investigation.

mally, this requires at least the equivalent of a procedure call for each generated event instance. This procedure call involves accumulating the attributes and invoking the packaging and delivery routines.

If procedure calling is the implementation technique, an event template

(event-class a_1 ... *time location).*

corresponds to an event generating annotation

`ebbaPostEvent` (*event-descriptor*, a_1, ..., *time*, *location*)

where the *event-descriptor* contains the information on how to package a representative instance of the *event-class* using the attribute values are bound by the actual call arguments. Another implementation technique would involve inserting trap instructions into a program at the appropriate places. As execution proceeds, the annotations trap to a code segment that accumulates the attribute values and calls the packaging and delivery library. A third technique would involve trap instructions signalling an event generation process that runs alongside the instrumented program.

Each of these techniques has good and bad features. All are functionally equivalent. They differ visibly in how the attribute values are accumulated and the packaging routines are called. The proper context for evaluating the merits of each involves three interrelated items

insertion ease Which determines how difficult it is to add a probe to the instrumented program. Important influences here include the time at which the annotation must be bound to the place in the program where it executes and the degree to which the insertion disrupts the flow of the annotated program.

access to attributes Once the annotation begins to execute, access to values that are to be bound to event attributes should be simple and uniformly available.

efficiency Since packaging and delivery are constant for all annotation techniques, savings is accrued by minimizing probe the effects due to the execution of the annotation.

Directly adding procedure calls to the program requires the annotations to be bound at the time the component is compiled. This is a very inflexible way of doing business. In order to shift or merge viewpoints the program must be recompiled or relinked with annotated components. Or, techniques must be used to ignore unwanted annotations–which still incur execution overhead (albeit minimal). Procedure calling, in spite of its drawbacks, is *the* easiest method of inserting annotations if source text is available (optimizing compilers could cause havoc with the assumed annotation placement). This is 100% of the reason that this technique is employed in the current toolset. Procedure calling has the important advantage that attribute values are readily accessible and can be bound with minimal overhead. Modifications to language compilers to insert event

generation text can effectively automate the procedure calling technique[7] (and allow optimized programs to be reliably instrumented).

Hardware trap instructions intended for debugging poorly serve their purpose because their execution is effectively little more than a jump through an interrupt vector. In use, a breakpoint trap generally replaces the instruction opcode at the place the transfer is desired. Executing the replaced instruction usually involves putting the original opcode back, adding a breakpoint to the succeeding instruction (or setting a single step flag in the processor status word). At this second transfer of control, the original breakpoint again replaces the intended opcode, so the program breaks the next time around. This second break, required to re-annotate the code properly, is inefficient and its effects better served if the return from interrupt could execute the replaced operation.

The greatest advantage of using trap instructions as a method of initiating primitive event generation is that they permit the annotations to be added as late as *during* execution of a component. Servicing the initiating trap instruction is clumsy because of the need to execute the instruction replaced by the trap. Access to attribute values must be managed by interpreting location information obtained from existing symbol tables, sometimes using system calls to cross process boundaries (e.g. Unix *ptrace(2)*). This technique has the advantage that if the source code is not available, it is still possible to add instrumentation annotations. Trap instructions that signal an associated monitor process (this is how UNIX *dbx(1)* and its varients work) have a disadvantage in that access to attribute values must be through a system interface.

Event Instance Output Formats.
The tuple form of an event instance is the invarient required by the event monitoring tools. Actual tuple format is not critical since the monitoring tools can be adapted to any type of packaged event instance representation. Currently all event instances are encoded into ascii text strings. This is largely a convenience for testing purposes and simplified integration of heterogeneous systems in the local operating environment. The primitive event descriptor that is included in an annotated component contains a *printf()*-like formatting string that guides event instance formatting. Most of the *libEBBA* components that are added to an executable program go to support this encoding. At the other end, the event abstraction component, decoding these strings is the major slow-down in servicing large volumes of incoming events. Using binary (unencoded) formats would considerably improve performance and consequently lessen the probe effects.

Specification-Executable Program Correlation.
An annoying feature of the current implementation is that design-level information (e.g. program defined variable names) is not available to the modelling process. Good examples of this are evident

[7] We have used this technique for a restricted event set.

in the MSG program. Early in the investigation the synchronization behavior of the program was suspect. These behaviors are implemented in terms of various lock variables used for synchronization and characterized by the events e_LockAcquired, e_LockPreset, and e_LockReleased. Each of these event classes contains a *lockname* attribute that uniquely identifies the lock variable that is affected. This variable is expressed as the value of the memory location that holds the lock. However, the view presented to the user is also in these terms–rather than symbolic terms readily available from symbol table information attached to the executable program image. Explicit mention of bindings for relevent tags or names to attributes values will require a more extensive Event Definition Language or a more capable user interface to direct these correlations.

A more difficult problem is the dynamic binding of underspecified design level information to the available values. This is evident in the process identifier attribute that is attached to many events. In the preceeding sections names like WORKER-1 and INITIAL were attached to the processes. These names are nowhere to be found in the design, they are purely mneumonics–they might as well be named KING, PAWN-1, etc. The inability to apply names when designing a behavioral model required certain events to be added to models, simply to supply a process identifier for filtering.

6. Conclusions

The tracking of this programmming problem is a quintessential example of using *EBBA* to solve a debugging problem. *EBBA* allowed the tool user to investigate the problem in a high-level, top-down manner, explain why hypotheses about the incorrect behavior were valid or not, and suggest what corrections needed be made to the software. The user was able to concentrate on *what* the program was doing rather than *how* it was doing whatever it was doing. There are two reasonable conclusions that might be drawn from this example: the *EBBA* technique works as advertised, and instrumentation will continue to be a source of difficulty.

The Technique Works As Advertised
The *EBBA* approach to debugging promotes understanding the causes of errors through behavioral observation and modelling, top-down detail-as-needed investigation, and shifting observational viewpoints to illuminate behavior from differing or merged perspectives. All of these techniques were employed to investigate the error provided by this program. Each shift in perspective was driven by information obtained from the previously visited viewpoint. Models developed from one viewpoint reinforced and augmented models created to explain other behaviors.

The Mandelbrot Set Generator program is composed of the components layered as outlined in Figure 17.

mandel this is the framework that holds the variables and implements the coarse control flow for

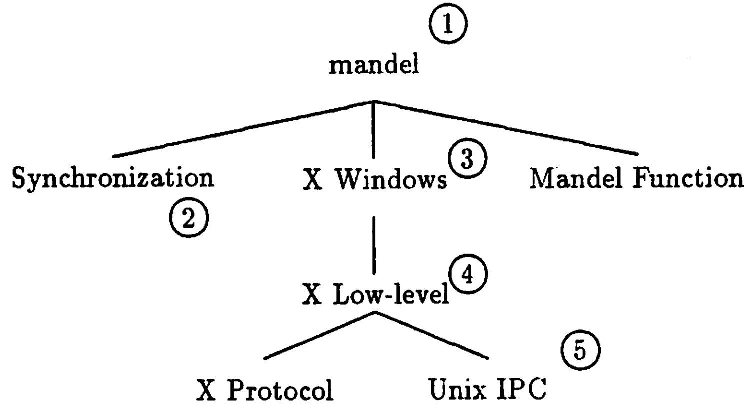

Figure 17: Mandelbrot Set Generator component layering

an overall program solution. This level is the highest behavioral level and is the conceptual level for the program. The mandel event library (Section 2.2) characterizes the behavior of this component.

mandel function is the code responsible for evaluating the points in the selected region of the complex plane. It is identified as the `FillRegions()` and `FillPatch()` routines. This component is characterized by the `e_TakePatch` and `e_DisplayPatch` events in the mandel library.

X Windows is responsible for communicating with the workstation display server. Its behavior is shown by the Xlib event library.

Synchronization keeps the components from colliding as they use the resources held by the organizing unit, mandel. Instrumentation of Sequent Parallel Programming Library routines is handled by the locks and microtask event libraries,

X Low-Level is responsible for managing the connection between the client and window server. The behavior at this level consists of structure editing and buffer filling. No event library directly characterizes this level. Some of its behavior can be illuminated by the generic procedure calling event library, EntXit. Further behavioral characterization would require events at the structure reference and assignment level.

X Protocol forms the rules for message exchange observed by the client and window server. Currently no event library exists to characterize this behavioral level.

Unix IPC is our old friend that in this case moves the X protocol requests and replys to and from the server.

The initial reaction to the problem was that the critical sections used to synchronize access to shared structures were the root of the fault. The investigation of the error proceeded from the top-level

mandel viewpoint ① down to the Synchronization component ②. A simple behavioral model from the Synchronization viewpoint provided evidence that the error was not a simple problem with the critical sections and it helped direct the investigation to a different area. A shift in viewpoint to X Windows ③ provided the visiblity necessary to demonstrate the behavior that caused the error. Observing the system through a model for the INITIAL process illuminated the behavior that was in error. At this level the suggestion for correcting the program is apparent; the INITIAL process should observe the `Xlock` protocol. A programming assumption based on ignorance regarding implications of the X Windows interface implementation was the direct cause of the error. However, it could be argued that only the error symptoms would be treated by this correction. A narrowing of the focus to include the X Low-level viewpoint ④ explained the class of behaviors that would cause this kind of error and suggested how the X Windows interface library might be changed to eliminate these types of errors.

Instrumentation Will Continue to Cause Difficulty

Instrumentation will not be a problem because of its timing or space requirements. Space is perhaps irrelevant and should not be triffled with any longer. Time is far more important, but with adequate compensation for its effects and choosing alternative viewpoints on a software component when time effects are critical, the probe effect can be effectively dealt with.

Adding the instrumentation necessary to provide behavioral visibility is still a bit of an art. Primitive event descriptor text and other aids can be created mechanically but little other help is currently available. We have had some limited success with automatic probe insertion at compile time and these techniques will become more sophisticated as more experience is gained. An important aspect of instrumentation is that once a software component has been instrumented, it and its behavioral characterization may be used over and over again. High-level models developed through the viewpoint describing the software component can also be reused as needed.

Perhaps the most difficult, ongoing problem with instrumentation is the policy and the ease with which probes are inserted into the software. Policy must deal with uncertainty about what behavioral characterization is actually presented by a set of primitive events. Characterization of component behaviors is conceptually easy, but does a particular primitive event or viewpoint actually capture that behavior? There is no proof or litmus test that may be applied that will give 100 percent confidence in an event/behavior characterization.

As sketched in Section 5.3 there are many ways to actually get the probes into software that needs to be monitored. Much of the difficulty with probe effects, policy, accurate characterization, etc. would be alleviated if it was easy to insert and remove probes as needed. Adding probes should be treated much like an oscilloscope probe used by hardware engineers. To effect this, two attitude changes must be made in the way "core" software is developed. First, language compilers

must become more open, that is, they should be written to allow diverse kinds of instrumentation to be added to the code they generate. Currently, the most widely used compilers include symbol table information to be used with some kind of a symbolic debugging tool. This is obviously a very rigid viewpoint that perpetuates an increasingly incapable debugging paradigm. Debugging tool aid should be expanded to include ways to insert annotations of an arbitrary but circumscribed nature, such as for *EBBA* or IPS-2 [5]. This would encourage a variety of techniques, each suited to particular types of programming problems.

The second attitude change is that heavily used reusable software components should be designed with probes in place that can be activated when needed. Language compilers and run-time environments can aid this process so it might be greatly aided by the above compiler changes. The envisioned embedded instrumentation should be integrated with its surroundings and hence be more sophisticated than simply surrounding instrumentation code with *if-then-else* brackets.

REFERENCES

[1] P.C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts/Amherst, January 1986.

[2] P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22, May 1988. Madison, Wisconsin.

[3] P.C. Bates and J.C. Wileden. High Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *Journal of Systems and Software*, 3(4), 1983.

[4] B.B. Mandelbrot. *The Fractal Geometry of Nature.* W.H. Freeman, 1982.

[5] B.P Miller, M. Clark, S. Kierstead, Sek-See Lim, and T. Torzewski. Ips-2: The Second Generation of a Parallel Program Measurement System. Technical Report 783, University of Wisconson-Madison, August 1988.

[6] H.-O. Peitgen and P.H. Richter. *The Beauty of Fractals.* Springer-Verlag, 1986. QA447.P45 86-3917.