MDAT: A Multithreading Debugging and Testing Tool

Eric Larson
Seattle University
901 12th Avenue
Seattle, WA 98122
(206) 296-5513
elarson@seattleu.edu

Rochelle Palting
Seattle University
901 12th Avenue
Seattle, WA 98122
rcpalting@gmail.com

ABSTRACT
MDAT is a multithreaded testing and debugging tool designed for students learning to program with multiple threads. MDAT automatically generates random schedules to allow students to more thoroughly test their programs. The design of MDAT takes full control over the scheduling allowing a failing run to be reproduced. To assist debugging, MDAT includes an output trace that shows the status of all threads, locks, and semaphores in the program and has an interactive mode that allows students to try out their own schedules. MDAT was effective at detecting deadlock and mutual exclusion violations in student submissions of the unisex restroom problem.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent programming.
D.2.5 [**Software Engineering**]: Testing and Debugging – *debugging aids, testing tools.*
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education.*

General Terms
Reliability, Verification.

Keywords
concurrency, synchronization, multithreading, testing, debugging

1. INTRODUCTION
One of the primary challenges of developing multithreaded applications is testing and debugging, especially for students who are first learning how to program with multiple threads. Testing and debugging are particularly important when designing programs that use synchronization primitives, such as locks and semaphores, as their programs may suffer from race conditions or deadlock. The paper presents MDAT – a multithreaded testing and debugging tool designed for students learning to program with multiple threads.

Students are often given problems such as the unisex restroom problem: *There is a single restroom that can be used by both men and women but not at the same time. There is no limit to how many people can be in the restroom at once but they all have to be the same gender.* This problem serves as a running example throughout the paper. There are two versions of this problem: the initial version as described above and the restricted version which is the same as the initial version with the added restriction of having at most four people in the restroom at once.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'13, March 6-9, 2013, Denver, Colorado, USA.
Copyright © 2013 ACM 978-1-4503-1868-6/13/03…$15.00.

It is difficult to test for race conditions and deadlock as they are often dependent on how the operating system schedules the individual threads. As a result, it is necessary to execute the same test with the same inputs multiple times to test different operating system schedules. This is a confusing concept to comprehend for students who are used to deterministic results each time a test is run. A further complication is that the schedules exercised by the operating system are very unlikely to be very aggressive in terms of finding race conditions. Since the first programs written by students are often small, the program rarely gets interrupted due to the expiration of a time slice. Instead the program runs until it encounters a system call (such as acquiring a lock or printing something to the screen). This means that it is highly unlikely for a program to get interrupted in the middle of a series of statements such as this:

```
countFemale++;
if (countFemale == 1)
  sem_wait(&male);
```

A common mistake for beginners is to not protect shared variables such as `countFemale` with a lock. However, a race condition may only surface if the operating system interrupts the thread after the if statement but before the wait statement (such that another thread can alter `countFemale` making the condition false). As noted earlier, the small size of these programs makes it highly unlikely that the thread will get interrupted in between those statements even for long-running programs.

Our MDAT tool alleviates these testing challenges in two ways. First, MDAT instruments the program by placing calls to invoke the scheduler after each statement allowing for more varied and aggressive schedules. Second, MDAT can be run to automatically test the program using a user-controlled number of random schedules. This permits students to easily test their programs several times without manual intervention.

If an error does occur in a multithreaded program, it can be difficult to debug. First, the bug may be dependent on a particular schedule and it may be difficult to reproduce that schedule. In addition, using a debugger to examine the state of different threads, locks, and semaphores can be intimidating to students who are new to multithreading programming.

MDAT addresses these concerns by having full control of scheduling. The tool is designed such that only one thread executes at a time so the underlying operating system has no decisions to make on what threads to execute. This permits the schedules produced by MDAT to be fully reproducible. When schedules are randomly generated, a random seed is displayed. The user can generate the exact same schedule using that random seed. In addition, MDAT creates a trace that displays the status of each thread, lock, and semaphore after each scheduler invocation.

Using this trace, a programmer can follow the steps that led to a particular problem.

For further testing and debugging, MDAT features an interactive mode that allows students to select which threads to run while displaying the program status on the screen. This mode encourages students to try different scenarios within their programs. More importantly, students can simply play around with their program increasing their comprehension on how multithreaded programs and synchronization primitives behave.

MDAT also has several benefits for the instructor. Using the interactive mode, MDAT can be used as an instructional aid in classroom presentations to demonstrate how locks and semaphores work. The interactive mode could also be used in a lab setting where students are given a buggy program and must devise a schedule that exposes the bug. MDAT can also help with evaluating student work. The instructor can run MDAT on student submissions to see if MDAT can find any errors. However, manual inspection of the submissions is still necessary since not every schedule is run. Exhaustively running all possible schedules is not computationally feasible.

2. MDAT OVERVIEW

The architecture of MDAT is shown in Figure 1. MDAT is designed for UNIX-based systems and programs written in the C programming language using the pthreads threading library.

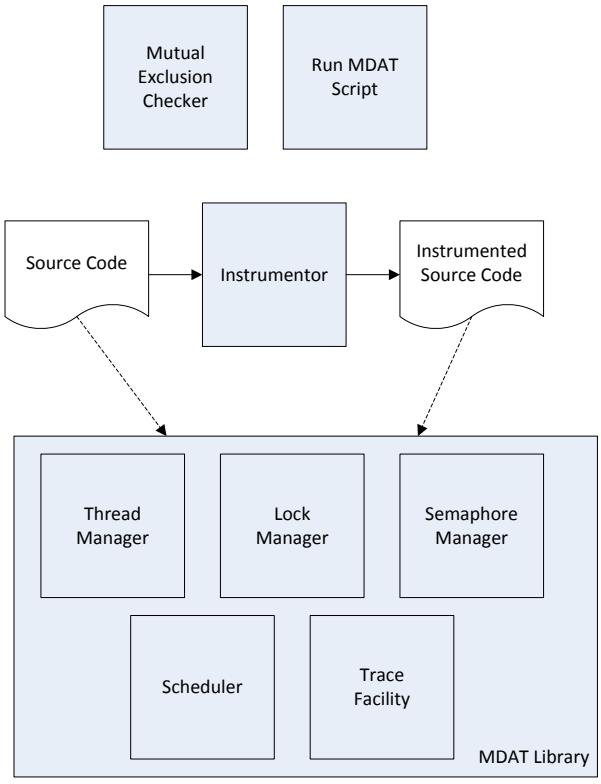

Figure 1. MDAT Architecture

The primary component of MDAT is the MDAT library which contains replacements for each of the lock and semaphore functions. The separation of the MDAT library permits it to be used with existing multithread applications with minimal effort. When writing a multithreaded program using MDAT, a student would use the corresponding MDAT functions instead of the normal pthreads functions commonly used in UNIX as shown in Table 1. To maximize realism, the argument lists for the MDAT functions are identical to those of the pthreads except for `mdat_mutex_init` and `mdat_sem_init`. These functions have an additional string argument that serves as the name of that entity that is subsequently used in the trace.

Table 1. MDAT Lock and Semaphore Functions

Pthreads function name	*MDAT function name*
`pthread_mutex_init`	`mdat_mutex_init`
`pthread_mutex_lock`	`mdat_mutex_lock`
`pthread_mutex_unlock`	`mdat_mutex_unlock`
`sem_init`	`mdat_sem_init`
`sem_wait`	`mdat_sem_wait`
`sem_post`	`mdat_sem_post`

MDAT only supports locks and semaphores as synchronization primitives. Furthermore, it only supports the basic operations shown in Table 1. Other operations such as `pthread_mutex_trylock` are not supported. This decision reflects what we covered in the class. In particular, only the basic lock and semaphore operations are taught. This choice is consistent with operating system textbooks which often only use these basic operations when introducing synchronization using locks and semaphores.

The MDAT library interface contains four additional functions:

- `mdat_init`: Initializes the MDAT library.
- `mdat_thread_start`: Registers a thread. Must be called by each thread at the beginning.
- `mdat_thread_finish`: Notifies MDAT that the thread has finished and not to schedule the thread any longer. Must be called by each thread once it completes.
- `mdat_invoke_scheduler`: Forces MDAT to invoke the scheduler and select a thread to run out of the threads that are ready (including the currently executing thread). In random mode, it will select a thread at random. In interactive mode, a prompt will be given to the user.

The calls to `mdat_invoke_scheduler` are automatically added by the *Instrumentor*, not by the programmer. The instrumentor adds a call to `mdat_invoke_scheduler` after each statement in the program. Furthermore, the instrumentor splits complex statements into two or more basic statements adding a call to `mdat_invoke_scheduler` after each one. This allows the tool to catch some, but not all, concurrency issues that arise if students incorrectly assume each statement is atomic. Splitting the statements further is left as future work.

The instrumentor only needs to be run on files that contain code that can be run concurrently. Optionally, a list of functions can be supplied to the instrumentor such that it will only instrument the functions specified on the list. To improve the debugging capabilities, `mdat_invoke_scheduler` has a location argument, a unique integer that is assigned by the instrumentor to each call it adds. As shown in Section 3, the location appears in the trace allowing the user to see where each thread is currently executing in the program. In addition to `mdat_invoke_scheduler` calls, the scheduler is invoked at the end of lock, unlock, wait, and post operations.

Internally, the MDAT library contains the following:

- *Scheduler*: The scheduler permits only one thread to be executed at a time. It also detects deadlock if no threads are able to run.
- *Thread Manager*: Manages all of the threads. Keeps tracks of which threads are ready to run and which threads are waiting due to a lock or semaphore.
- *Lock Manager*: Manages the locks. Keeps track of which thread holds each lock and which threads, if any, are waiting for each of the locks.
- *Semaphore Manager*: Manages the semaphores. Stores the current values of each semaphore and which threads, if any, are waiting for a post operation.
- *Trace Facility*: Responsible for displaying important events and the current state in the trace file.

The scheduler is implemented using signals. When a thread is not executing, it calls `sigwait` which causes the thread to wait until it receives the Linux user-defined signal SIGUSR1. When a thread switch occurs, the thread that is currently active sends the next thread to execute the SIGUSR1 signal using `pthread_kill`. To avoid issues at start up, all threads must call `mdat_thread_start` to register with MDAT. All threads must register before execution can start. Once all the threads have registered, the thread selected to start (chosen randomly or by the user in interactive mode) will begin execution. All other threads will wait for their turns using `sigwait`. A thread must call `mdat_thread_finish` when it completes. This lets the scheduler know not to schedule that particular thread anymore.

The thread manager, lock manager, and semaphore manager are essentially tables that keep track of the threads, locks, and semaphores respectively. The various operations update these tables accordingly. An important field in the thread manager is the current state of each thread: running, ready, waiting for a lock, waiting for a semaphore, or completed. The scheduler will only select from threads that are running or ready. If no thread is running or ready, the scheduler will report deadlock and abort the program. The state of each lock (locked or unlocked) is tracked by the lock manager. If a thread needs to wait due to a lock that is already acquired by a different thread, the state of the thread is updated and the scheduler is invoked to choose a different thread. The semaphore manager works in a similar fashion.

While the MDAT library can check for deadlock, it cannot check for mutual exclusion violations as these are specific to the programming task. The *Mutual Exclusion Checker* checks for these types of violations relying on output statements that are produced by the program. The mutual exclusion checker will need to be modified based on the mutual exclusion requirements of the program. For example, a checker for the reader / writer problem would ensure that a writer has exclusive access in the critical section. The checker implementation for the unisex restroom problem is described in Section 4.

Finally, the *Run MDAT Script* is a Python script that will run the program a user-specified number of times. This allows students to test their programs on as many random schedules as they desire. Once an error is detected, either due to deadlock or a mutual exclusion violation, the script will stop. The random seed for each run is displayed, and the trace files of the failing run are saved so students are able to reproduce the failing run.

3. USER INTERFACE

To use MDAT, simply compile the instrumented source code produced by the instrumentor and link with the MDAT library. We altered our source code such that the MDAT configuration could be controlled using command line arguments. For instance, one command line switch directs MDAT to run in interactive mode. The command line arguments are parsed and passed into MDAT using `mdat_init`.

An excerpt of a trace is shown in Figure 2. For consistency, the trace and interactive outputs are identical except when interactive mode prompts the user for a thread. The first three lines are informational messages. Messages are displayed any time a thread is switched or a synchronization function is executed. The status table that follows is divided into three sections: threads, locks, and semaphores. For each thread, the current location is given – the unique identifier of the last `mdat_invoke_scheduler` call executed by that thread. Looking at the first line of Figure 2, thread 6 executed `mdat_invoke_scheduler` with unique identifier 7 and this is reflected in the status table that follows. Programmers, using the instrumented source code, can follow along using this location field.

Within the thread section of the status table, the status column shows the current state of the thread. It can be one of the following states: ready, running, waiting-lock, waiting-sem, or completed. If the thread is waiting, the name of the lock or semaphore it is waiting on is displayed in the last column.

The lock section displays the name of each lock, the status of the lock (which includes the thread currently holding the lock if it is held), and a list of threads waiting for that lock. The semaphore section is similar. It displays the name of the semaphore, its current value, and a list of threads waiting for that semaphore. We chose to use a semaphore implementation that decrements for every wait operation allowing the value to be negative. For example, `genderBlock` in Figure 2 has a value of -1.

When running in interactive mode, a prompt appears anytime there is a choice regarding threads. At the start, the user is prompted on which thread should start running first. Each time the scheduler is invoked, the user is prompted on which thread to run next. Only the threads that are ready are acceptable; the user is asked to select a different thread if their selected thread is waiting or has already completed. If threads are waiting, the user is also prompted on which thread to wake up when a lock is released or a semaphore is incremented. Referring back to Figure 2, thread 2 releases lock `maleMutex`. MDAT randomly chose to wake up thread 4 but it could have selected threads 0, 8 or 10. In interactive mode, the user would have been given the choice to wake up either thread 0, 4, 8, or 10. Note that selecting a thread to wake up is independent of the choice of which thread to schedule next. Even though the lock was transferred to thread 4, thread 6 was selected to run next.

```
[Call 7] Switching from thread 6 to thread 2
Thread 2 is releasing lock maleMutex
Transferring lock maleMutex from thread 2 to thread 4
********************************************************************************
|THREAD ID   |LOCATION    |STATUS       |WAITING ON                             |
|0           |7           |waiting-lock |maleMutex                              |
|1           |3           |waiting-lock |femaleMutex                            |
|2           |9           |running      |                                       |
|3           |21          |completed    |                                       |
|4           |17          |ready        |                                       |
|5           |6           |waiting-sem  |genderBlock                            |
|6           |7           |ready        |                                       |
|7           |21          |completed    |                                       |
|8           |7           |waiting-lock |maleMutex                              |
|9           |21          |completed    |                                       |
|10          |17          |waiting-lock |maleMutex                              |
--------------------------------------------------------------------------------
|LOCK NAME                |STATUS       |WAITING THREADS                        |
|femaleMutex              |held by 5    |1                                      |
|maleMutex                |held by 4    |0 8 10                                 |
--------------------------------------------------------------------------------
|SEMAPHORE NAME           |VALUE        |WAITING THREADS                        |
|occupancy                |2            |                                       |
|genderBlock              |-1           |5                                      |
********************************************************************************
[mdat-mutex-unlock] Switching from thread 2 to thread 6
Thread 6 tried to acquired held lock maleMutex - thread 6 must wait
********************************************************************************
|THREAD ID   |LOCATION    |STATUS       |WAITING ON                             |
|0           |7           |waiting-lock |maleMutex                              |
|1           |3           |waiting-lock |femaleMutex                            |
|2           |9           |ready        |                                       |
|3           |21          |completed    |                                       |
|4           |17          |ready        |                                       |
|5           |6           |waiting-sem  |genderBlock                            |
|6           |7           |waiting-lock |maleMutex                              |
|7           |21          |completed    |                                       |
|8           |7           |waiting-lock |maleMutex                              |
|9           |21          |completed    |                                       |
|10          |17          |waiting-lock |maleMutex                              |
--------------------------------------------------------------------------------
|LOCK NAME                |STATUS       |WAITING THREADS                        |
|femaleMutex              |held by 5    |1                                      |
|maleMutex                |held by 4    |0 6 8 10                               |
--------------------------------------------------------------------------------
|SEMAPHORE NAME           |VALUE        |WAITING THREADS                        |
|occupancy                |2            |                                       |
|genderBlock              |-1           |5                                      |
********************************************************************************
```

Figure 2. Example of MDAT trace file

4. UNISEX RESTROOM IMPLEMENTATION

The unisex restroom problem can be structured like many classic synchronization problems and divided into four sections:

- Entry section (person is entering the restroom)
- Critical section (person is using the restroom)
- Exit section (person is leaving the restroom)
- Remainder section (person is outside drinking more water)

The critical and remainder sections are actually uninteresting in enforcing mutual exclusion – all of the work is done in the entry and exit sections. The students were given these four functions. Each of the functions started with an output statement that logged the start of the function. The resulting log was then used by the checker. Students could only add code to the entry and exit sections after the output statement. Students also had to add code to a fifth function that initialized shared variables, locks, and semaphores. These five functions are resided in a standalone file.

Students were also given a driver file that is responsible for setting up the threads. Each thread executes the four sections in order for a user-specified number of times. The driver file can be reused for other programs and the classical synchronization problems that are set up using these four sections.

The mutual exclusion checker is invoked after a run has concluded and uses the log to determine if a mutual exclusion violation has occurred. The log consists of entries that describe which section each thread has entered along with the gender of that thread. To enforce the mutual exclusion rule, the checker uses two counters: one for the number of males in the restroom and one for the number of females. The checker scans the log entries in order. When someone enters the critical section, the appropriate counter is incremented. When someone leaves the critical section, the appropriate counter is decremented. Mutual exclusion occurs if both counters are non-zero at the same time.

Students were also asked to write a second implementation to the restricted version of the unisex restroom problem such that at most four people could be in the restroom at once. To support this restriction, the checker was modified to flag an error if either gender's counter is greater than four. A command line switch was also added to the checker so this check could be turned off or on depending on which version of the problem was being worked on.

5. RESULTS

To assess the potential impact of MDAT, we used the student submissions of the unisex restroom problem that were assigned in the Fall 2011 and Fall 2012 offerings of the first author's Operating Systems and Networks class. Students had to provide solutions for both the initial problem and the more restrictive version. Solutions did not need to ensure fairness or lack starvation but they could not be overly restrictive (such as allowing only one person in the restroom at once).

In the Fall 2011 offering, students did ***not*** have access to the MDAT library at the time; they had to test and debug their programs themselves manually. However, as part of their assignment, students had to create a mutual exclusion checker that they could use to help test their programs. The assignments were graded manually looking for mutual exclusion violations and possible cases of deadlock. Once MDAT was completed (which occurred well after the completion of the course), 21 submissions were converted so they could run with MDAT.

In the Fall 2012 offering, students did have access to MDAT. There were 28 submissions for the assignment. To reduce copying from the previous offering, the problem was changed to an identical problem with a different situation: prohibiting old children and young children from simultaneously being on a play structure at once. For clarity, the remainder of this section is written in the context of the unisex bathroom problem. Students were also given a survey at the end of the assignment to provide feedback on their experience using MDAT.

We ran the submissions from both quarters using MDAT for 200 different randomly-generated schedules with 12 threads and 10 rounds per run. The results are shown in Table 2.

Table 2. Results of using MDAT on Student Submissions

MDAT reported...	Fall 2011 (w/o MDAT)	Fall 2012 (MDAT)
No errors (correct solution)	6	23
No errors (incorrect solution)	0	1
Deadlock (all 200 runs)	10	0
Deadlock (some runs)	1	0
Mutual exclusion violation (all 200 runs)	1	1
Mutual exclusion violation (some runs)	0	2
Both deadlock & mutual exclusion	2	1
Hung due to incorrect spin loop	1	0

The fact that 15 of the 21 submissions are incorrect in the Fall 2011 offering clearly demonstrate the need for such a tool. While using MDAT, only 5 of the 28 submissions were incorrect in the Fall 2012 offering. MDAT was able to detect four of the five faulty submissions. In the incorrect submission that was not detected by MDAT, the synchronization was too restrictive in that it only allowed one person to use the restroom at once. MDAT does not catch this particular problem but the mutual exclusion checker could be modified to warn the user of this situation.

In addition, in virtually all of the cases, very few runs are needed to catch the error. This implies the aggressive scheduling with more opportunities to switch threads has a significant impact on finding multithreaded problems. The only exception to this are the two solutions from Fall 2012 that encountered mutual exclusion violations on only some of the runs. In both cases, mutual exclusion violations were only detected in 4 of the 200 runs.

For the Fall 2011 offering, we compared the results returned from MDAT with our manual grading of the assignments. Quite embarrassingly, the manual grading of these assignments only caught 6 of the 14 deadlock cases. Manual grading did properly detect the three mutual exclusion violations and deemed the six solutions with no errors to be correct.

Now consider the restricted version of the problem where the restroom has a maximum capacity of four people. Solutions that got the initial part wrong were not considered further since the corresponding solution to the restricted version of the problem was incorrect for the same initial reason. Of the 29 solutions that remained, MDAT did not find any errors. Of these 29 solutions, 27 solutions correctly implemented the restriction. Two solutions (both from the Fall 2012 offering) were incorrect in that it was possible to have more than four people use the restroom but MDAT did not generate a random schedule for this scenario.

To address the lack of data, we used a solution of the initial unrestricted version of the problem with the more restrictive mutual exclusion checker. Since the solution lacks a mechanism for restricting capacity, it is capable of allowing more than four people in the restroom. The solution was run 10,000 times. Only 1 of the 10,000 runs reported a case where four people were in the restroom at once. In this particular situation, using a randomized scheduler was not effective.

The students in the Fall 2012 offering were asked a variety of questions regarding their experience using MDAT. Here is the set of YES / NO questions and their responses:

- Q1. Did MDAT detect deadlock? *24 YES, 2 NO*
- Q2. Did MDAT detect a case where men and women used the restroom at the same time? *16 YES, 9 NO*
- Q3. Did MDAT detect a case where more than four people used the restroom at once? *8 YES, 16 NO*
- Q4. Did you look at the output trace? *22 YES, 4 NO*
- Q5. Did you use interactive mode? *17 YES, 9 NO*

Considering the number of correct solutions that were ultimately received, MDAT was successful in detecting errors. We found it surprising that eight students encountered an error with more than four people using the restroom at once given how rare it was encountered when evaluating the final submissions.

Students who answered yes to Q4 were asked to gauge the usefulness of the output trace on a five point scale from 1 (very useless) to 5 (very useful). They were also asked to provide comments on the trace. The same two questions were posed to those who answered yes to Q5 asking them to assess the usefulness of the interactive mode. The results of the two multiple choice questions are listed in Table 3.

Table 3. Usefulness of MDAT Features

Usefulness of ...	*(Very Useless)*				*(Very Useful)*
	1	*2*	*3*	*4*	*5*
output trace	1	3	7	4	7
interative mode	0	2	0	7	7

The usefulness of the output trace was mixed. While some students found the trace to be useful, several felt that the trace was hard to follow, very dense, and a lot of data to scroll through. One comment that really stood out was "Once I understood what I was looking at it was useful, but that took a long time to do."

Based on Table 3, the students who used interactive mode found it to be useful. Looking at the comments, some students used interactive mode to get a better understanding of how context switching, locks, and semaphores worked. Other students found interactive mode to be very helpful in debugging and testing their implementations. One student noted that using output statements he/she added to the code was more helpful.

Lastly, students were asked what improvements should be made to MDAT. Several students suggested on reducing the level of detail on the trace output. Another common suggestion was to better correlate the position on the trace to the source code and to the errors reported by the mutual exclusion checker. In particular, it was difficult to find an error detected by the mutual exclusion checker in the trace.

6. RELATED WORK

The design of MDAT was inspired from CHESS [1]. CHESS is a multithreaded testing tool that systematically generates different schedules for testing, an improvement over simply randomly generating schedules. CHESS works on Win32 and .NET platforms. Inspect [8] is a similar tool but works on C programs with pthreads. ConTest [4] and CalFuzzer [6] generate multithreaded tests for Java. ConTest biases the scheduler towards obtaining multithreaded coverage. CalFuzzer focuses on potential concurrency bugs that were flagged by different tools. All four of these tools are designed for real-world programs where MDAT is focused on assisting students with multithreaded programming.

From an education perspective, Bi and Biedler [2] developed a Java thread visualization tool that helps students see graphically the states of the different threads over time. Ricken and Cartwright [7] propose using a test-first approach to writing concurrent programs in Java. They use a specially designed unit testing framework that overcomes problems with using traditional unit testing frameworks. Currently, their testing is dependent on the schedule produced by the JVM but they plan to incorporate a technique that systematically generates schedules based on heuristics in the future. Bruce et al. [3] propose introducing concurrency as a topic in the first computer science course. They initially select examples that avoid or minimize race conditions but eventually introduce the topic later in the course. A more thorough treatment of testing and debugging is left for future courses. Gopalakrishnan et al. [5] give an extensive list of resources for teaching concurrency and describe how they use Inspect [8] and CHESS [1] in their course.

7. CONCLUSION AND FUTURE WORK

MDAT is a multithreaded testing and debugging tool designed for students learning to program with multiple threads. For the unisex restroom problem, it was effective in helping students find deadlock and mutual exclusion violations where men and women used the restroom at the same time. It was not effective at finding violations of the added restriction that only four people could use the restroom at once. While many students found MDAT to be useful for testing and debugging, some students felt that the output trace was too long and difficult to parse.

For future work, the first obvious step is to improve the tracing facility so it is easier to understand. One enhancement is to have the mutual exclusion checker be integrated into MDAT instead of a separate entity. This will allow state associated with mutual exclusion state to be part of the trace making it easier to debug. Another enhancement is to create a GUI with the ability to collapse and expand the trace. Initially the trace would contain a description of the different events that occurred and the user could click on an event to see the various tables at that point. This would allow the user to find where the error is taking place and then only investigate the state near where the error occurred.

Another direction of future work is to improve the ability to catch concurrency bugs such as the restriction of having four people in the restroom at once. Are there improvements to the scheduler to catch this particular type of bug? Another source of bugs is forgetting that individual statements cannot be assumed to be atomic. By breaking individual statements into smaller pieces, the instrumentor will be able to insert more scheduler invocations in the middle of individual statements. The downside of more scheduler invocations is that the resulting trace is longer.

Lastly, the MDAT interface can be adjusted to use the actual locking and semaphore functions. This would allow the student to write their program that would compile and execute normally with or without MDAT. This would make the exercise seem more real and demonstrate the benefit of creating testing and debugging tools. It would also permit students to use MDAT on other concurrent programs.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. We also thank the students in the Fall 2012 offering of Operating Systems and Networks at Seattle University for providing feedback on MDAT.

REFERENCES

[1] Ball, T., Burckhardt, S., de Halleux, P., Musuvathi, M., and Qadeer, S. 2011. Predictable and Progressive Testing of Multithreaded Code. *IEEE Software* (May/June 2011), 77-83.

[2] Bi, Y., and Beidler, J. 2007. A Visual Tool for Teaching Multithreading in Java. *Journal of Computing Sciences in Colleges* 22, 6 (Jun. 2007), 156-163.

[3] Bruce, K., Danyluk, A., and Murtagh, T. 2010. Introducing Concurrency in CS 1. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Mar. 2010), 224-228.

[4] Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby G., and Ur, S. 2003. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience* 15 (2003), 485-499.

[5] Gopalakrishnan, G., Yang, Y., Vakkalanka, S., Vo, A., Aananthakrishnan, S., Szubzda, G., Sawaya, G., Williams, J., Sharma, S., DeLisi, M., and Atzeni, S. 2009. Some resources for testing concurrency. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (Jul. 2009).

[6] Joshi, P., Naik, M., Park, C., and Sen, K. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Jun. 2009), 675-681.

[7] Ricken, M., and Cartwright, R. 2010. Test-First Java Concurrency for the Classroom. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (Mar. 2010), 219-223.

[8] Yang, Y., Chen, X., and Gopalakrishnan, G. 2008. *Inspect: A Runtime Model Checker for Multithread C Programs.* Technical Report UUCS-08-004. University of Utah.