Some resources for teaching concurrency *

Ganesh Gopalakrishnan, Yu Yang, Sarvani Vakkalanka, Anh Vo,
Sriram Aananthakrishnan, Grzegorz Szubzda, Geof Sawaya,
Jason Williams, Subodh Sharma, Michael DeLisi, Simone Atzeni
School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA
{ganesh}@cs.utah.edu

ABSTRACT

With the increasing emphasis on exploiting concurrency efficiently and correctly, the lack of suitable pedagogical material for teaching concurrency is a growing problem. In this paper, we summarize a recently concluded class as well as some independent projects in the area of concurrency and multi-core computing that offer some insights to address this problem. We examine background papers, the teaching of low level concurrency, and the teaching of threading and message passing. The use of dynamic formal verification tools in a class setting is discussed in some detail. We conclude with a summary of pedagogical material being assembled, including exercises from a popular textbook on MPI solved using our dynamic verifier ISP. Our observation is that the teaching of concurrency is greatly facilitated by the use of dynamic push-button formal verification tools that can handle non-trivial concurrent programs. Given the growing number of publications on how to teach concurrency as well as employ new programming approaches, our work addresses the somewhat neglected topic of using modern dynamic formal verification methods within the context of widely used concurrency approaches and libraries.

Categories and Subject Descriptors

D.1.3 [**Programming techniques**]: Concurrent Programming – Threading, Message Passing, Memory Models; K.3.2 [**Computers and Education**]: Computer science education – Teaching Concurrency; D.2.4 [**Software Engineering**]: Formal Methods – Dynamic Verification.

General Terms

Verification, Concurrency, Education

Keywords

Multi-core, MPI, Pthreads, Memory Models, Dynamic Verification, Computer Science Education

*Supported in part by Microsoft, NSF CNS-0509379, CCF-0811429, and SRC Contract No. 1847.001

PADTAD'09, July 19–20, 2009, Chicago, Illinois, USA

1. INTRODUCTION

The quest to exploit concurrency at all levels – from hardware through applications – has made it imperative that future curricula incorporate concurrency at all levels of the education pipeline. The magnitude of this task is easily appreciated if one takes stock of planned hardware/software offerings by the industry, on one hand, and the state of our curricula that is still geared towards sequential programming, on the other. It is no surprise that the hardware and software industry has realized the need to close this gap, as evidenced by their various academic programs emphasizing concurrency education (*e.g.*, [1, 2]).

This paper summarizes our initial efforts to incorporate concurrency into a graduate level class taught by the first author, with help from his co-authors, It also draws from a few complementary projects aimed at evaluating research tools and available textbooks. As this topic is so diverse, we take the approach of first summarizing how the course and the projects actually evolved. We then summarize some specific resources we would like to offer to the broader community in the form of tools and pedagogical examples. This paper barely scratches the surface of this vast and important topic: the reader is requested to take this paper as the start of a much needed discussion.

The tools used in our class were the following: Inspect [46, 47, 48], CHESS [34, 35] (for threading), and ISP [26, 29, 30, 31, 32, 33] for message passing. The choice of Inspect was mainly for the reason that it is the only model checker that can handle large (relatively speaking) concurrent POSIX threading based C programs and verify them using dynamic partial order reduction, yielding significant interleaving reduction. For a given input test harness, Inspect verifies a program for safety properties including data races, deadlocks, and assertion violations. The choice of ISP was mainly for the reason that it is the only model checker that can handle large MPI C programs. For a given input test harness, ISP verifies a program for safety properties including communication races, deadlocks, assertion violations, and resource (*e.g.*, MPI object) leaks. In ISP, we employ our own customized dynamic partial order reduction algorithm called POE [29] that has proved to be indispensable in practice. Also, both Inspect and ISP are developed in our group. ISP is in a fairly polished state, and is being downloaded worldwide [26]. Inspect is also available for downloads. The choice of CHESS was to offer a perspective on how to apply formal methods to .NET applications. We have not explored

tools such as the Java Pathfinder [36] that allow Java programs to be directly handled. These would be included in future studies. We now summarize some of the course highlights, followed by a road map of this paper.

Use of Dynamic Verification Tools: We found that it is quite advantageous to teach concurrency using formal dynamic verification tools [41] such as Inspect, CHESS, and ISP. Students who do not have a formal methods background are drawn by the debugger-like user interface provided by these tools. The fact that we have integrated ISP into a visually appealing graphical user interface (Microsoft Visual Studio for now, with Eclipse planned for later) proved to enhance the appeal of ISP. CHESS also has a powerful Visual Studio integration. Using such tools along with their interfaces, we found that it was relatively easy to show students the risks of bug omissions during conventional testing and the benefits of guaranteed coverage provided by dynamic formal verification (an early compilation of such results appears in [25]). Students also realized that the reduction of redundant concurrent thread/process interleavings made possible by algorithms such as dynamic partial order reduction (DPOR) [40] (used in Inspect) and preemption bounding (used in CHESS) are far more reliable than *ad hoc* interleaving bounding methods employed during conventional testing. As far as our verifier ISP goes, it includes an MPI-specific version of DPOR called POE [29]. We would go so far as to say that without interleaving reduction techniques, concurrent program analysis and verification would be impossible to apply in practice.

Use of Model based Verification Tools: In previous classes with concurrency emphasis that we have taught, our tool of choice was SPIN [43]. Likewise, without ISP, one would employ tools such as MPI-SPIN [44] for verifying MPI programs. These approaches fall under the heading *model based verification*. While model based verification is advantageous when it comes to teaching concurrency at a more conceptual level, given the limited amount of time in modern curricula, we would strongly suggest the use of dynamic verification tools. They eliminate the need for model building, which can be a deterrent for students, as it is tedious as well as error-prone. Given the choice between conventional testing and model building that we offered in some of our previous classes, there was a pronounced inclination on part of students toward testing. With dynamic verification tools, the entry barrier is removed, with the tools proving to be even more helpful in most cases than conventional testing tools.

Currently, there are many efforts in furthering dynamic verification (e.g. [45, 37]). We believe that many more such efforts are urgently needed in order to build critical mass in this area. Given that debugging concurrent software is very expensive in practice [42], we must convince real designers to begin using verification tools on their source codes. Besides, concurrent software of the future will most likely be written using languages that call external libraries and APIs. To handle such programs, a whole range of dynamic formal verification techniques must be explored. While we did not include static analysis tools in our studies, static analysis approaches that produce few or no false alarms would be welcome additions to the concurrency tool suite. In addition to scaling well, they can also complement dynamic verification. Unfortunately, most static analysis techniques have not fared well in these areas, especially when dealing with real-world concurrency APIs and languages.

Choice of Textbooks/papers: While there are many excellent textbooks that can help teach concurrency with particular relevance to future multi-core computing (e.g., [3, 4]), they do not discuss how to apply formal analysis techniques to the source codes they present. As we discuss in several places in this paper, modern dynamic verification tools offer a concrete path towards closing this gap. At present, we are applying dynamic verification methods to the examples in [3] and are building a resource page for users of this textbook.

The remainder of the paper is organized as follows. Section 2 details our syllabus. Section 3 illustrates how we taught the topic of shared memory consistency models. Section 4 discusses how we taught threading, and Section 5 discusses how we taught message passing. We discuss work in progress in applying ISP to do all the exercises from a popular textbook on message passing (Section 5.1). When finished, this exercise may offer a unique teaching resource for message passing. Section 6 has our concluding remarks.

2. A PLAUSIBLE MULTI-CORE COMPUTING SYLLABUS

Our syllabus was refined all along, and now looks as follows:

- Provide an overview of current topics and discussions occurring around multi-core computing.
- Provide hands-on experience on parallelization and potential speed-up.
- Provide exposure to the basic concurrency paradigms of threading and message passing.
- Provide a detailed introduction to the topic of memory models.
- Provide more detailed exposure to the use of dynamic formal verification tools for threading and message passing.
- Provide an overview of writing state transition semantics using labeled transition systems. Ask them to hand-calculate the outcome of running a short MPI program using a simple operational semantic definition.
- Help students firm-up independent project selections.
- Use [3] and cover many topics, including: correctness criteria, various locking protocols, memory models, foundations of atomic objects, linearizability, the concept of consensus numbers, etc.

This syllabus reflects the first author's background as well as his focus on correctness issues. That said, it did help us cover a variety of topics not found in a single textbook. We now detail what was covered under each of the above topics.

Overview of current topics: Sutter [8], and Sutter and Larus [9] provide high level motivations for multi-core computing. A good overview of the importance of addressing heterogeneous multi-core CPUs is provided in [7]. This paper is accompanied by a spreadsheet that allows students to vary the assumptions about core sizes and the portion of

code that is serial, and obtain speed-up curves. The prevailing consensus is that future multi-core systems will employ heterogeneous cores for reasons supported by this analysis.

The paper by Dennis [5] provides a historical perspective on concurrency, and the virtues of employing a functional/declarative notation. Similar points are also made by Blelloch [6] who has developed a successful pedagogical approach based on functional programming. Cantrill et al. [10] take a somewhat sober look at the multi-core hoopla, claiming that the advances in parallel software design over the decades of the 70s and 80s created the need for multi-core CPUs of today. We finished off this section with an in-depth look at a recent 'E-book' on multi-core computing produced by Leiserson and Mirman [11] that provides a comprehensive overview of work, span, and parallel speed-up.

Hands-on Experience: We then provided hands-on experience for students in terms of using a parallel machine and observing/understanding speed-up. We found that the most expedient way to do this was to employ the Cilk system [23]. Students with minimal background can begin using Cilk and observed speed-up on examples such as various versions of the merge sort, parallel array sum, etc. When we did this work, we only had access to classical symmetric multiprocessor (SMP) clusters that most institutions have. Our most capable machine had eight dual-core CPUs. Also we installed the "classical Cilk" from MIT [12], as the industrial version Cilk++ was not available at the outset. Nevertheless this segment of experience, and the reading of Cilk papers was found to be very fulfilling by most students. From knowing nothing about concurrency, they emerged with a very good awareness of the basics plus acquired significant hands-on experience – all in a matter of weeks.

Introduction to Threading and Message Passing: We found the on-line tutorials produced by Barney [13] to be a very good introduction to several concurrency paradigms, including threading, message passing using MPI, as well as many other paradigms. We applied our tools Inspect and ISP to verify many of their examples for a collection of standard safety properties. This exercise proved to be a valuable learning aid to our students – both in becoming familiar with Pthreads and MPI, in learning the use of dynamic verification tools, and in detecting documented bugs in these examples.

3. TEACHING MEMORY MODELS

We now turn to a topic that is considered too esoteric by many people. We show that by choosing simple examples, this topic can indeed be made very accessible and 'gut-level.' All the material discussed in this section can be covered in two assignments in about two weeks. We divide our presentation into language level memory models (Section 3.1), architectural memory models (Section 3.2), and the use of memory ordering checking tools (Section 3.3). We present all these topics as given out in our assignments. We skip a formal presentation of these topics which have been covered in many other places.

3.1 Language Level Memory Models

Most compilers (e.g., `gcc`) are unaware of what is "safe" to do in a multithreaded or multi-core situation. To study this, let us experiment with `interlock1.c` below.

```
#include <pthread.h>
int p1=0, p2=0;
char aa=0, bb=0; // handshake bits
void * thread_routine1(void * arg)
{ do {
    aa  = 1; while(!bb); bb = 0; p1++;
    if (!(p1 % 10))
      printf("progress1\n");
  } while(1);
}
void * thread_routine2(void * arg)
{ do {
    bb  = 1; while(!aa); aa = 0; p2++;
    if (!(p2 % 10))
      printf("progress2\n");
  } while (1);
}
int main()
{ pthread_t  t1, t2;  pthread_create(&t1, 0, thread_routine1, 0);
  pthread_create(&t2, 0, thread_routine2, 0);  pthread_join(t1, 0);
  pthread_join(t2, 0);  return 0;
}
```

Now, the students were asked to answer these questions:

gcc, no optimization: Do `gcc -o interlock1 interlock1.c -lpthread` and run `interlock1` on a uniprocessor and then a multiprocessor machine. Did you see the execution "hang"? Explain in a few sentences.

gcc, with optimization: Include the -O3 flag. Now what are the results you obtain? (It must hang!). Explain in a few sentences as to why. Will it hang on a uniprocessor? (Answer: No!)

The students could appreciate that due to the *store/load* optimization not being done, the unoptimized versions did not hang, while the optimized versions did. They were encouraged to do `gcc -S interlock1.c` and then look at the assembly code emitted in `interlock1.s`. They could see that the store and load instructions were permuted, thus leading to the threads being able to hang.

3.2 Architectural Memory Models

The students were now shown how the unoptimized code itself would break on a multiprocessor. However, they often had to re-run the following code multiple times, to run into a situation where the underlying hardware actually ends up reordering instructions. (Often times, they had to repeatedly abort the execution through control-C and re-run the code – basically to trick the underlying Operating System to trigger a condition of hardware re-ordering!) If they were persistent, they could get the `Caught violation` message to get printed! Again, the take-away message was that without a deeper understanding of shared memory consistency models, one is likely to sow bugs that are very difficult to reproduce.

```
#include <pthread.h>
int aa=0, bb=0, zz=0;
int c1=0, c2=0;
void * thread_routine1(void * arg)
{ do {
    c1++; aa = 1; if (!bb) zz++; zz = 0; bb = 0;
  } while(1);
}
void * thread_routine2(void * arg)
{ do {
    c2++; bb = 1; if (!aa) zz++; zz = 0; aa = 0;
  } while (1);
}
void * thread_routine3(void * arg)
{ while(1)
  if (zz > 1)
```

```
  printf("Caught violation, c1=%d, c2=%d, zz=%d\n", c1, c2, zz);
}
int main()
{ pthread_t  t1, t2, t3;
  pthread_create(&t3, 0, thread_routine3, 0);
  pthread_create(&t1, 0, thread_routine1, 0);
  pthread_create(&t2, 0, thread_routine2, 0);
  pthread_join(t1, 0);  pthread_join(t2, 0);
  pthread_join(t3, 0);
  return 0;
}
```

Next, the students were asked to put in an `mfence` instruction [50] into the assembly file to prevent the store to load reordering. They tried – in vain – to trigger the violation! In our experience, such hands-on experiences get the students to understand what memory models are basically about.

3.3 Teaching Tools for Memory Models

In this segment, they were given a demo of a SAT-based memory consistency checker called `mpec` [14, 15]. They were also asked to run ARCHTEST – an execution based calibrator of actual multi-processor machines [16] – on our lab machines. We then asked the students to study the reference document on the x86 memory model [17], asking them to re-state the memory model – as best as possible – in the parlance of Collier's classifications. While such exercises are likely to receive different answers, they at least allow the students to help apply what they learn in one context to another. These were followed by the reading of the papers by Boehm [19, 20] which further reinforced the need for rigorous concurrency specifications pertaining to compilation. A particularly delectable 'treat' for the students was the program in [19] where threads exploited races to compute Prime numbers faster. Students coded this example both using C/Pthreads (without using locks!) and Cilk (by causing deliberate races!)

4. TEACHING THREADING

Students were now taught the basic use of the Inspect tool and model check the actual Pthreads/C code of a (buggy) producer/consumer routine using it [22]. To the best of our knowledge, Inspect is the only publicly available tool that employs DPOR to model check Pthreads/C programs of this type. We also encouraged some students to verify a `C#` implementation of this code using CHESS. The fact that such programs can be push-button verified using Inspect provides the much sought-after 'instant gratification' which can help convey the value of dynamic formal verification to the skeptical student. We also built an Emacs editor-based user interface which helped students step through Inspect-generated error traces. While both Inspect and CHESS can suffer from interleaving explosion, we could still compare these tools by using the following criteria (valid at the time of writing): (i) soundness - Inspect is sound for a given test harness. CHESS is also sound for all practical purposes, as it can search fully up to a target pre-emption bound (a pre-emption bound of 3 is usually adequate); (ii) race detection - Inspect does race detection, while CHESS will have this feature in the next release, (iii) liveness - CHESS supports a pragmatic version of liveness while Inspect does not currently have a notion of liveness.

5. TEACHING MESSAGE PASSING

A large body of material has been assembled to teach message passing using MPI. Here we summarize the material:

- Various examples on MPI programming have been assembled at [27], with many of them illuminating the examples from the tutorial available from [13].
- Parts of a matrix multiplication challenge - originally proposed by Siegel [28]. These have been solved using ISP.
- The same matrix multiplication examples were run on our local cluster machine and the speedups were plotted [28].

All in all, these examples capture the abilities of ISP as well as its graphical user interface. The references cited here show how one can employ ISP to understand and debug MPI programs. (Note: This part of our work was the result of an independent project.)

5.1 Pedagogical Material from Pacheco's MPI Textbook

One exercise we are engaged in is *solving as many of the exercises as we can* from the popular book on MPI programming, by Pacheco. The current website representing our work is at [18]. We believe that once this project is finished, anyone wanting to teach MPI using this textbook will find ISP to be a ready companion. (The author of [21] was recently contacted by us; he was very appreciative of ISP's bug-hunting abilities.) The take-away message from our experience in this area is that a considerable amount of work remains to be done in integrating usable formal methods into textbooks being written on concurrent programming. However, with the recent successes in developing dynamic formal methods for Java, `C#`, .NET, MPI, and Pthreads that have been discussed so far in this paper, the formal methods community is in a position to collaborate with practitioners who may write future textbooks on concurrency APIs, and provide for the readers of these books usable formal verification tools.

A 'Red Herring: A strange bug that we encountered in solving examples from Pacheco's book is now briefly summarized. This is an eye-opener – again serving to remind us how many subtle bugs one may end up facing in the 'concurrency land.' In one exercise, an MPI process is to receive from a process whose rank is calculated as `(e-1)%2`. In one instance, `e` happened to be `0`, and so the rank turned out to be `-1%2` which C promptly evaluates to be `-1`. Now, `MPI_PROC_NULL` was defined to be `-1` in the MPI system. The MPI standard further says that receiving from `MPI_PROC_NULL` is tantamount to a `no op`! Therefore, all said and done, in this example, the MPI receive was rendered a `no op` due to a C "feature." (Mathematicians may regard `-1%2` as `1`, in which case the receive would not end up being a no op.) In reality, what happened is that this triggered a bug in ISP which was not designed to support MPI receives from a null process. This bug has, since then, been fixed in ISP. ISP now deadlocks on this example because, essentially we are losing one 'MPI receive' due to this "C feature." The moral of this story is that it takes a lot of domain knowledge before one can debug concurrent programs.

As members of the formal verification community, it would be good to have the time resources to expend on understanding and dealing with these situations, given the lim-

ited "publication value" that such examples may carry. If we remain excessively swayed by the sheer novelty of our ideas at the risk of neglecting to cater to such "realities," we may end up designing tools that do not help today's practitioners. This aspect may be addressed by emphasizing, in one's publication portfolio, a good balance between theoretically deep papers and also "community service" papers that cater to the detailed needs of the concurrent programming community.

6. CONCLUDING REMARKS

We described a class in progress where several approaches to verify concurrent programs and to understand concurrency situations are being taught. This is just one instructor's approach to make up a syllabus as the class went on. However, it is also unclear how else to teach such a class in such a rapidly evolving area. In the end, we were pleased that we took a hands-on approach so that the students obtained a firm grounding on many slippery concurrency notions. The students firmed up their learning by choosing many ambitious class projects including: (i) writing a Raytracer using Cilk++; (ii) verifying the non-blocking queues from [3] using CHESS; (iii) implementing a work-stealing algorithm in Pthreads/C and verifying using Inspect; (iv) several other projects involving the study of various concurrency APIs, and their multi-core realizations.

All in all, this paper presents a detailed experience report with citations that may prove to be useful to others. If there is any definite advice to impart, it is to use tools and testing experiments well before theoretical topics are taught. For instance, our syllabus in Section 2 shows that we are teaching the harder conceptual material from [3] only after many hands-on exercises have been done. In our opinion, this will ensure that the material is appreciated in the right context by students who may otherwise be overwhelmed and/or think that the concurrency notions are disconnected with reality.

Acknowledgements: The first author likes to especially thank the students of CS 5966/6966 of Spring 2009 for their patience and feedback through his class.

7. REFERENCES

[1] Intel Academic Alliance http://software.intel.com/en-us/articles/courseware-access/

[2] External Research, Microsoft. http://research.microsoft.com/en-us/collaboration/

[3] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan-Kauffman, 2004.

[4] Lawrence Snyder and Calvin Lin. Principles of Parallel Programming Addison-Wesley, 2008.

[5] Jack B. Dennis. Toward the computer utility http://csg.csail.mit.edu/Users/dennis/essay.htm

[6] Guy Blelloch. Parallel Thinking http://www.cs.cmu.edu/~blelloch

[7] M. Hill and M. Marty, Amdahl's Law in the Multicore Era IEEE Computer, July 2008. http://www.cs.wisc.edu/multifacet/amdahl

[8] Herb Sutter. "The Free Lunch is Over" http://www.gotw.ca/publications/concurrency-ddj.htm

[9] Herb Sutter and James Larus. Software and the concurrency revolution. http://portal.acm.org/citation.cfm?id=1095421

[10] B. Cantrill. Real-world Concurrency. CACM Volume 51, Number 1, 2008. http://portal.acm.org/citation.cfm?doid=1400214.1400227

[11] Charles Leiserson and Ilya Mirman, How to survive the multicore software revolution. http://www.cilk.com/ebook/download5643

[12] The MIT Cilk System. http://supertech.csail.mit.edu/cilk/

[13] Using LLNL's Supercomputers. https://computing.llnl.gov/tutorials/agenda/index.html

[14] Ganesh Gopalakrishnan, Yu Yang, Hemanthkumar Sivaraj. QB or Not QB: An Efficient Execution Verification Tool for Memory Orderings. Computer Aided Verification 2004, 401-413, LNCS 3113.

[15] MPEC: A SAT-based checker for Itanium Executions. http://www.cs.utah.edu/formal_verification/software/mpec.

[16] W. W. Collier. Reasoning about Parallel Architectures. Prentice-Hall, 1992.

[17] Sections 7.2 and 7.3, Intel Software Developer's Manual, Chapter 7. : http://www.intel.com/design/processor/manuals/253668.pdf

[18] Geof Sawaya. Examples from Pacheco's book pacheco. http://www.cs.utah.edu/formal_verification/geof/pacheco/table.html.

[19] Hans Boehm. Threads cannot be implemented as a library. http://www.hpl.hp.com/techreports/2004/HPL-2004-209.html PLDI 2005.

[20] Hans Boehm. Reordering Constraints for Pthread-style Locks. http://www.hpl.hp.com/techreports/2005/HPL-2005-217R1.html?jumpid=reg_R1002_USEN PPoPP 2007.

[21] Peter S. Pacheco. Parallel Programming with MPI. Morgan-Kauffman, 1997.

[22] Pthreads/C code of a producer/consumer routine. http://www.eng.utah.edu/~cs5966/Week4/prodcons.c

[23] The Cilk++ tool suite. http://www.cilk.com

[24] http://www.eng.utah.edu/~cs5966/

[25] http://www.cs.utah.edu/formal_verification/ISP_Tests/.

[26] http://www.cs.utah.edu/formal_verification/ISP-release/.

[27] Examples of Using ISP on the LLNL benchmarks, and the Red-Black benchmarks. http://www.cs.utah.edu/~jtwilla/WORK/ISPTests.html

[28] Files containing Message Passing pedagogical examples. http://www.cs.utah.edu/formal_verification/padtad09-files/

[29] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. Computer Aided Verification 2008, 66-79, LNCS 5123.

[30] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. *PPoPP 2009.*, 261-269.

[31] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. A formal approach to detect functionally irrelevant barriers in MPI programs. *EuroPVM/MPI 2008.* 265-273, LNCS 5205.

[32] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. *PADTAD-VI 2008.*

[33] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp. Implementing efficient dynamic formal verification methods for MPI programs. *EuroPVM/MPI 2008.* 248-256, LNCS 5205.

[34] `http://research.microsoft.com/en-us/projects/chess/`.

[35] Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Pages 446–455, 2007.

[36] `http://javapathfinder.sourceforge.net/`.

[37] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, Univ. of California, Berkeley, Sep 2008.

[38] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.

[39] M. Dwyer, J. Hatcliff, and D. Schmidt. Bandera : Tools for automated reasoning about software system behavior. In *ERCIM News, 36*, Jan. 1999.

[40] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

[41] P. Godefroid, B. Hanmer, and L. Jagadeesan. Systematic software testing using VeriSoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal*, 3(2), April-June 1998.

[42] P. Godefroind and N. Nagappan. Concurrency at microsoft - an exploratory survey, 2008. EC2 (Exploiting Concurrency Efficiently and Correctly), Princeton, 2008. `http://www.cs.utah.edu/ec2/2008`.

[43] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[44] S. F. Siegel and G. S. Avrunin. Verification of MPI-based software for scientific computation. *SPIN 2004.*

[45] J. Yang, et al. MODIST: Transparent Model Checking of Unmodified Distributed System. NSDI 09. To appear.

[46] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software. SPIN 2007, Pages 58-75, Springer LNCS 4595.

[47] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. SPIN 2008, Pages 288-305, LNCS 5156.

[48] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic Discovery of Transition Symmetry in Multithreaded Programs using Dynamic Analysis. SPIN 2009, Accepted.

[49] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/

[50] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufman, 2004.