Tiddle: A Trace Description Language for Generating Concurrent Benchmarks to Test Dynamic Analyses

Caitlin Sadowski
supertri@cs.ucsc.edu

Jaeheon Yi
jaeheon@cs.ucsc.edu

Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Dynamic analysis is a promising technique for finding concurrency bugs in multithreaded programs. However, testing a dynamic analysis tool can be difficult. Researchers end up writing large amounts of small benchmark programs. Since the benchmarks themselves are concurrent programs, they may execute nondeterministically, complicating testing of the analysis tool.

We propose testing dynamic analyses by writing traces in a simple trace description language, Tiddle. Our implementation, written in Haskell, generates deterministic multithreaded Java programs for testing dynamic analyses. We report that it is substantially easier to write programs with incriminating bugs such as race conditions in Tiddle than the corresponding Java source code version, reducing the amount of source code to maintain and understand. Although our implementation is targeted towards Java, the ideas extend to any other languages which support mutable fields and multiple threads.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs – *specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *program analysis*

General Terms

Languages, Reliability, Verification

Keywords

Traces, race conditions, atomicity violations, concurrency, dynamic analysis

WODA '09, July 20, 2009, Chicago, Illinois, USA.

1. INTRODUCTION

Concurrency bugs are insidious and hard to reproduce. They appear at the whim of the system, when a particular problematic sequence of operations occurs. Since they are scheduling dependent, testing is not always effective: buggy test cases will execute correctly sometimes. An alternative solution is to use program analysis techniques to isolate general purpose concurrency bugs like data races or atomicity violations.

Program analysis techniques fall into two main camps: static or dynamic. Static analyses analyze the source code. They tend to produce many false positives (*i.e.* are imprecise) because they do not know dynamically available unification information. A *program trace* or *interleaving* is the sequence of operations executed by a run of a program. Dynamic analysis tools instrument a program, and then analyze traces produced by the instrumented program at runtime to find bugs in the program being executed. These tools are typically not sound (*i.e.* produce false negatives) because they are based on a particular trace, which may not be indicative of all possible program behaviours. Some hybrid tools straddle both camps; they do a post-mortem analysis on program traces, or combine static and dynamic analysis in some other framework.

This paper is targeted towards researchers who write dynamic analysis tools to find concurrency bugs in multithreaded applications. Our dynamic analysis framework is written in Java, and so that is the language we target. However, the implementation could be easily extended to other languages which support concurrency and mutable fields (*e.g.*, C++); this would involve some pretty printing changes in the module which translates Tiddle ASTs to Java ASTs.

Many dynamic analysis tools exist to discover concurrency errors in programs, such as data races [39, 38, 30, 46, 11, 32, 15], atomicity violations [42, 16, 45, 43, 27, 17, 14], and deterministic parallelism violations [36]. The tools are often intricate concurrent programs in their own right. Testing dynamic analyses involves writing many small programs that exhibit a fault (or do not exhibit a fault) to check if the analysis correctly runs. However, since these tests must be multithreaded, they may not be deterministic! One (inadequate) method of attempting to force a deterministic schedule is to pepper `yield()` or `sleep()` statements throughout the tests; this does not guarantee determinism, even if the statements are placed properly. The tests must also contain a lot of boilerplate code for setting up multiple threads. In short, they are both tedious and error-prone.

What is a better way of quickly testing dynamic analy-

ses? It is our observation that small *traces* exhibiting certain faults are a natural way of describing test cases. Writing small traces on the whiteboard as an aid to discussion is natural. We have created a trace description language, Tiddle, that captures basic concurrency notions in execution traces. We translate traces into *deterministic* Java source code for the purpose of testing dynamic analyses. This guarantees that the same execution trace results from every run of the program. We are able to achieve this determinism using a key assumption: that the dynamic analysis framework can optionally ignore specified method calls.

We have implemented a compiler, written in Haskell, which translates Tiddle trace statements into Java source code. The language and implementation is extensible, allowing customization to suit the needs of the user. We also use Tiddle to generate traces that exhibit the same behaviour as the source trace, so as to verify that a dynamic analysis behaves as expected on equivalent traces.

Our contributions are thus:

1. A language-agnostic domain-specific language (DSL) for describing trace behaviour, Tiddle.
2. A compiler, implemented in Haskell, that translates Tiddle traces into deterministic concurrent Java programs.
3. An experience report about the usefulness of Tiddle.

2. RELATED WORK

Describing Program Traces.

Tiddle is a language for describing program execution traces. There has been some previous work on describing program traces for various purposes. A formal look at trace models of Prolog programs is explored in [22] for building high-level Prolog debuggers. Several authors have looked at simplifying, organizing and abstracting stored traces [6, 3].

The Test Behaviour Language (TBL) for traces provides a concise way of describing and testing for trace properties [6]. Various temporal logics can express bug patterns [2] or other arbitrary properties [9], and runtime verification systems [7] may monitor traces for violations of patterns and properties. Tiddle could potentially be used to specify concurrency bug patterns and anti-patterns to look for in traces.

Some previous work uses traces to describe patterns. Generated traces that exhibit a statistical profile are used to test architecture performance characteristics [10]. These traces are not translated into programs, but instead run directly on a simulator. Traces are mined for patterns of procedure calls [19]. One could extend Tiddle to describe problematic data access patterns for atomic-set-serializability violations [18].

DSLs.

Tiddle is a simple DSL that captures the essence of execution traces: a well suited domain. DSLs help eliminate unnecessary boilerplate ("eliminate notational noise" [44]) and allow for concise, comprehensible programs. We have used the full language design approach [44] to better capture the practice of writing a trace on the whiteboard, and then translating that trace to a complete Java program. Similar in intent is Teapot [5], a domain specific language for writing cache coherence protocols. The Teapot system generates C code implementing a specified protocol, and has a verification system to check protocol designs. Related systems exist for generating code to implement security protocols in Java [33] or C# [23]. Although the intent of these systems is similar to Tiddle, the focus is different; Tiddle code is used to create test cases for dynamic analysis tools, while protocol generators are used to prototype new protocol designs.

Deterministic Replay.

There has been some prior work on *deterministic replay* of multithreaded Java applications [8, 35, 4, 40][1]. Essentially, these systems record information about thread scheduling so that a buggy execution trace can be replayed after a crash occurs. Without a replay system in place, it may be difficult to reproduce failures; different interleavings may not result in an error state. Conceptually, this work is similar to ours in that traces are used to run a Java program. The way traces are generated and used is very different in the two techniques. Replay captures traces from running programs; we construct traces for purposes of understanding our analysis tools. Replay systems restrict runtime behaviour of an existing program; Tiddle code compiles directly to deterministic Java source code.

Unit Testing Concurrent Software.

MultithreadedTC [34] is a framework that allows programmers to encode scheduling information inside of unit tests. An external clock is added to the tests, along with a thread responsible for clock maintenance and deadlock detection. Test snippets can block waiting for clock ticks or assert when a tick has happened; ticks occur when all threads in the system except the maintenance thread are blocked. The trace of operations is not made explicit in this system, and so it is possible to accidently generate nondeterministic tests. Additionally, if MultithreadedTC was used to create tests for a dynamic analysis tool, operations by the maintenance thread could confuse an analysis tool. MultithreadedTC also aims to eliminate boilerplate Java code (although not to the degree of Tiddle). MultithreadedTC was inspired by ConAn [26], a script based testing framework that also uses a clock ticks to enforce specific interleavings.

Concurrency Benchmarks for Research.

There has been a recent drive to create a benchmark of concurrency bugs, targeted at dynamic analyses [12, 13]. Eytani et al. claim that their benchmark has had an impact on the research community, and is now used by several groups. The concurrency bugs in the benchmark follow different bug patterns, but were mostly found in programs written by novices. In comparison, a Tiddle trace describes a run time event pattern, and represents a lower level of abstraction than the benchmark programs. Tiddle allows researchers to quickly and efficiently generate small and simple test cases for their property of interest.

Most similar to our work is TMUnit [20], in which a simple description language is used to generate workloads and test the semantics of transactional memories. The user specifies threads, transactions and their schedules to test a particular interleaving on a Software Transactional Memory (STM) system. The user can also specify invariants to be checked

[1]Other replay systems exist for Java [24, 41], but here we are focused on replay systems targeted at concurrency.

at each stage of the run. In contrast, specifications in Tiddle of whether or not the dynamic analysis flags an error exist at a level above the Tiddle program itself. Harmanci et al. validate their work by testing five STM systems. This work highlights the relevance of a trace DSL beyond dynamic analysis frameworks.

3. SEMANTICS OF MULTITHREADED PROGRAMS

We begin by describing a simple semantics of multithreaded program traces. Refer to [36] for a similar fully formalized operational semantics. A program consists of a number of concurrently executing threads that manipulate shared variables $x \in Var$ and locks $m \in Lock$. Each thread has a thread identifier $t \in Tid$. For simplicity, in our programming model variables are global.

3.1 Operations

A *trace* α captures an execution of a multithreaded program by listing the sequence of operations performed by the various threads. The set of operations that thread t can perform include:

- $rd(t, x, v)$ and $wr(t, x, v)$, which read and write a value v from variable x;
- $acq(t, m)$ and $rel(t, m)$, which acquire and release a lock m;
- $begin^l(t)$ and $end^l(t)$, which demarcate each `atomic` [17] or `deterministic` [36] block labelled l;
- $fork(t, u)$, which forks a new thread u;
- $join(t, u)$, which blocks until thread u terminates.

This lower level abstraction of traces allows us to ignore objects and their composition; more complicated concurrency bugs boil down to short sequences of these basic operations. We are concerned with the traces produced from an actual run of a source program. These traces will be *well-formed*; they fulfill expected constraints when forking, joining, acquiring and releasing. For example, every release operation has a corresponding acquire operation by the same thread earlier in the trace, and no operations by a thread u occur in a trace prior to the forking of thread u.

3.2 Conflicts

Two operations in a trace *conflict* if they satisfy one of the following four conditions:

- **Communication conflict**: they are by two different threads, access the same variable, and at least one of the accesses is a write.
- **Lock conflict:** they are by two different threads, and acquire or release the same lock.
- **Fork-join conflict**: one operation is $fork(t, u)$ or $join(t, u)$ and the other operation is by thread u.
- **Program order conflict**: they are performed by the same thread.

The *happens-before relation* $<_\alpha$ for a trace α is the smallest transitively-closed relation on operations in α such that if operation a occurs before b in α and a conflicts with b, then a *happens-before* b.

Two traces are *equivalent* if one can be obtained from the other by repeatedly swapping adjacent non-conflicting operations. Equivalent traces yield the same happens-before relation and exhibit equivalent behaviour. In general, dynamic analyses should act identically on equivalent traces.

3.3 Transactions

A *transaction* in a trace α is the sequence of operations executed by a thread t starting with a $begin^l(t)$ operation and containing all t operations up to and including a matching $end^l(t)$ operation. In some cases [36], the operations of all threads forked within a transaction are also included.

4. METHODOLOGY

4.1 Tiddle Grammar

We describe the grammar of Tiddle. A BNF-style representation is given in Figure 4.1.

```
trace ::= trace op
        | op
op ::= rd   Tid Var (Val)
     | wr   Tid Var (Val)
     | acq  Tid Lock
     | rel  Tid Lock
     | fork Tid Tid
     | join Tid Tid
     | beg  Tid Label
     | end  Tid Label
Tid ::= Int
Var ::= String
Val ::= Int
Label ::= String
Lock :: = String
```

Figure 1: Tiddle Grammar

In Tiddle, a trace is a list of operations. Each operation belongs to a thread, and the thread identifier is indicated by an integer in the first argument. Reads and writes specify a variable and optionally a value. If a read operation specifies a value, an `assert` statement is added to verify that the read returned the expected value; variables are initialized to 0 by default. Since the generated programs are deterministic, this is simply a check to make sure the trace is specified as desired. Acquires and releases specify a monitor lock. Forks and joins specify the thread identifier for the forked or joining thread. Begin and end operations demarcate atomic blocks, or transactions [16].

In Java, the reads and writes correspond to accesses to a static field. The acquires and releases correspond to a `synchronized` block. Forks and joins correspond to Java's `start()` and `join()` methods. Begin and end operations are Java blocks annotated as `atomic`. Alternatively, begin and end operations can denote method boundaries. This allows the generated programs to be checked with a general "all methods are atomic" specification. These Java programs represent simple, straightforward examples that embody the buggy pattern described by the corresponding Tiddle trace.

Data Races.

A data race occurs when two threads simultaneously access a shared variable, at least one access is a write, and there is no synchronization between the two accesses.

Tiddle operations support modeling data races. Our compiler generates working, complete Java code from a partial trace. There is no need to write out the full execution trace of a program; only the relevant lines of the trace need to be specified and other operations (*e.g.*, forking all the threads involved) are added automatically. Here is a data race specified in two lines (`x` is initially 0):

```
rd 1 x
wr 2 x 1
```

There is no explicit synchronization to prevent the trace from being reordered to:

```
wr 2 x 1
rd 1 x
```

so the final value of `x` – 0 or 1 – depends on the nondeterministic schedule of operations. Specifying this data race is only two lines of Tiddle code– but these two lines are translated to more than 50 lines of Java source code (Figure 2).

Atomicity Violations.

Atomicity is a general concurrency specification that isolates program behavior of code blocks. Informally, a code block is atomic if all executions of that code block have the same effect as if the code block executed in isolation from the rest of the program. The database community refers to this property as serializability. We can describe an atomicity violation in Tiddle:

```
beg 1 a
rd  1 x
wr  2 x
rd  1 x
end 1 a
```

The atomic block is indicated by the begin and end operations, and must be equivalent to a serial trace where the atomic block is executed without other threads interleaving. Here, the write to `x` in between two subsequent reads of `x` means that this trace is not equivalent to any serial execution of the atomic block. This trace compiles to about 70 lines of Java source code.

Note that the operations and their semantics reflect the kind of dynamic analyses we are interested in: detecting data races and atomicity violations. Tailoring the Tiddle language to operations for different dynamic analyses, such as detecting atomic-set-serializability violations [18] or new types of concurrency bugs should be straightforward.

4.2 Determinism and Synchronization

Race conditions and atomicity violations are nondeterministic by nature. A test program that has such bugs is difficult to use, because the error may manifest rarely and only under specific conditions. Thus, test programs should be deterministic – even those with nondeterministic bugs.

We use barrier synchronization to implement determinism for otherwise nondeterministic test programs. The test program generated from the specification trace executes only one operation per barrier. All threads in the program move

```
public class Test {
  static int x = 0;
  static CyclicBarrier cb = new CyclicBarrier(2);
  static CyclicBarrier cc = new CyclicBarrier(2);
  static int numThreads = 2;

  static public void await(CyclicBarrier c)
    throws BrokenBarrierException,
             InterruptedException {
    c.await();
  }

  public static void main(String[] args) {
    final Thread t2 = new Thread() {
      public void run() {
        try {
          int _z = 0; //for reads
          await(cc);
          await(cb);
          await(cc);
          x = 1;
          await(cb);
        } catch (InterruptedException e) {
          e.printStackTrace();
        } catch (BrokenBarrierException e) {
          e.printStackTrace();
        }
      }
    };
    final Thread t1 = new Thread() {
      public void run() {
        try {
          int _z = 0;
          await(cc);
          _z = x;
          await(cb);
          await(cc);
          await(cb);
        } catch (InterruptedException e) {
          e.printStackTrace();
        } catch (BrokenBarrierException e) {
          e.printStackTrace();
        }
      }
    };
    t1.start();
    t2.start();
  }
}
/* Input trace:
     rd 1 x
     wr 2 x 1
*/
```

Figure 2: Generated Java Code with Race

in lockstep from one barrier call (`await()`) to the next. Between each barrier call, precisely one thread executes an operation, while all other threads simply race to the next barrier and block.

We assume that dynamic analysis frameworks may elide certain method calls from being instrumented. We believe this is a reasonable assumption for any dynamic analysis framework, since deciding what to instrument depends on what analysis is being performed. With this capability to elide method calls, we ignore all calls to `await()` when analyzing the test program. The program continues to run deterministically; however, the analysis tool will not observe the barrier synchronization and correctly analyze the program. If barrier calls were not elided from the analysis, the test program will still run deterministically. However, the analysis tool may now emit false positives (or negatives) because the tool reacts to the observed contention on the barrier.

Forks and joins present interesting synchronization issues, because these operations change the number of threads racing to a barrier. We use `CyclicBarrier` from the Java standard library, which allows recycling the barrier between waits, although a `BrokenBarrierException` gets thrown if the number of threads to trip a barrier is changed while some threads are still blocking on that barrier. We solve this problem by using two cyclic barriers, so that the second of the two barriers can be reset to a new value while the other threads are all blocking on the first one.

Optionally, all barrier synchronization may be elided during code generation to produce *small nondeterministic* programs that exhibit a particular type of multithreaded bug. This way, Tiddle can be used for test generation in situations where ignoring `await()` calls is not possible. These could also be small test cases for testing static analysis tools. Alternatively, Tiddle can be configured to add `sleep()` or `yield()` statements to the (barrier-free) code, although the test programs will still be nondeterministic.

4.3 Code Generation

The compiler for the Tiddle language is written in Haskell [21]: a lazy, pure, strongly-typed functional language. Haskell has powerful constructs that make it easy to concisely define and manipulate abstract syntax trees (ASTs). We used Alex [29] and Happy [28] (Haskell versions of Lex and Yacc [25]) to generate a lexer and parser, respectively. The Haskell implementation generates a (custom) Java AST from the trace AST provided by the parser. Code generation is handled by a separate pretty printer [31] for the Java AST. Haskell is a terse language: our entire implementation totals about 300 lines of Haskell code. The functional style of Haskell make it a natural choice for AST generation and manipulation. We chose this language because of the combination of conciseness and expressivity, and because it was fun to use.

4.4 Equivalent Traces

In addition to generating the code for one particular trace, Tiddle can also generate all traces equivalent to a particular trace. We scan through the trace to find source operations which have only outgoing happens-before edges, and then recursively add all interleavings of operations respecting happens-before order. In essence, this pulls out all equivalent traces by exploring different paths through the nodes in the happens-before graph for a trace, only adding an operation when all prior operations which happen-before it have been added to the trace. A series of equivalent traces can be compiled into one (long) Java program, enabling a single run of the analysis to confirm that all equivalent traces find the same errors (or lack thereof).

5. A CASE STUDY

We discovered that *traces are a quick way of testing dynamic analyses*. The trace abstraction arises naturally when thinking about concurrent program execution. Two common tasks when developing a dynamic analysis is to write down interesting trace fragments on paper, and then to write code that exhibits this trace for testing a dynamic analysis implementation, or to mentally run through the analysis logic for such a trace. With Tiddle, the tedium of writing code to *test* a dynamic analysis is alleviated. Quick testing aids in the *development* of a dynamic analysis.

We also discovered that a trace is a very nice, concise way to *describe* many concurrency bugs. For example, the easiest way to explain a data race is to produce an example of one, and it is very easy to do just that with Tiddle.

We have used Tiddle as a development aide for dynamic analyses. As a concrete example, we have developed SideTrack [37], an atomicity violation analysis that looks for specific trace patterns, using Tiddle. During implementation, we discovered a bug in how SideTrack handled nested locking by a simple Tiddle trace. When we extended our analysis with another atomicity violation pattern, it was easy to test that the new pattern worked properly. Finally, when refactoring SideTrack's implementation, we used all the Tiddle traces as a regression test suite. In addition, we have used Tiddle to test the SingleTrack [36] deterministic parallelism checker. To date, 70 or so Tiddle programs have been written.

Our initial experiences have been very positive, and Tiddle has become an integral part of our workflow. By using Tiddle, the amount of code to maintain has dropped, since we may store traces instead of Java source files. If we think up a tricky test case, we just write it out in Tiddle, instead of trying to work through the example by hand.

6. FUTURE WORK

We are extending the Tiddle language to handle more Java synchronization constructs (for example, `wait` and `notify`) as needed. We are also implementing the ability to embed Java code snippets directly inside a Tiddle trace. This makes Tiddle more extensible at the expense of simplicity.

Another possible direction for future work is to create a smaller test program that replicates faults in the original program. We could record problematic traces of large programs, then instantiate a smaller program that exhibits the same multithreaded bug. This is similar to record-and-replay, but a smaller program can replay the original problem: perhaps better suited for prototyping dynamic analyses.

One way to take Tiddle to the next level would be to develop it into a formal specification language for bugs such as data races and atomicity violations. This specification language may serve as an easier way of understanding safety properties through violation specifications, and could be integrated into a runtime verification system like MOP [7].

We are interested in ways which Tiddle can be targeted

to help others in the program analysis community. We are in the process of releasing Tiddle as a Haskell package in HackageDB [1].

7. CONCLUSION

We introduce Tiddle, a simple DSL for specifying execution traces aimed at enabling testing of dynamic analyses. Our trace language provides an abstraction of a multithreaded program execution trace. The Tiddle compiler, written in Haskell, translates Tiddle traces into deterministic Java source code programs, or nondeterministic small tests that exhibit a particular type of multithreaded bug. This allows researchers to test the behaviour of an analysis tool on a particular trace, and easily generate small test cases. Tiddle can also be used to generate equivalent traces, to check for consistency of an analysis across traces with the same behaviour.

We have found Tiddle to be a valuable part of the development process for dynamic analysis tools. We aim to continue the dialogue on how dynamic analyses are tested and developed.

8. ACKNOWLEDGEMENTS

Special thanks to Stephen N. Freund for originally suggesting this project. Thanks to Cormac Flanagan for clarifying and expanding the ideas and Kenneth Knowles for assistance with Haskell. Thanks to Ian Pye for reading and commenting on a final draft of this paper, even though he was traveling at the time.

9. REFERENCES

[1] *Hackage*. http://hackage.haskell.org.

[2] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of Java applications for multithreaded antipatterns. In *International Workshop on Dynamic Analysis (WODA)*. ACM, 2005.

[3] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A framework for the efficient encoding of general trace data. In *ACM SIGPLAN-SIGSOFT Workshop on Program Anaylsis for Software Tools and Engineering (PASTE)*, Nov. 2002.

[4] R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

[5] S. Chandra, B. Richards, and J. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *Software Engineering, IEEE Transactions on*, 25(3):317–333, 1999.

[6] F. Chang and J. Ren. Validating system properties exhibited in execution traces. In *International Conference on Automated Software Engineering (ASE)*. ACM, 2007.

[7] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. *SIGPLAN Notices*, 42(10):569–588, 2007.

[8] J. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*. ACM, 1998.

[9] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. In *International Workshop on Dynamic Analysis (WODA)*, pages 1–7, New York, NY, USA, 2005. ACM.

[10] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2000.

[11] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–255, 2007.

[12] Y. Eytani, R. Tzoref, and S. Ur. Experience with a Concurrency Bugs Benchmark. In *IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW)*, 2008.

[13] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Parallel and Distributed Processing Symposium*, 2004.

[14] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, 2008.

[15] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009. To appear.

[16] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[17] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[18] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering*. ACM, 2008.

[19] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *International Workshop on Dynamic Analysis (WODA)*. ACM, 2003.

[20] D. Harmanci, P. Felber, V. Gramoli, and C. Fetzer. TMunit: Testing Transactional Memories. 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2009.

[21] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007.

[22] E. Jahier, M. Ducassé, and O. Ridoux. Specifying prolog trace models with a continuation semantics. In *LOPSTR: Selected Papers form the 10th International Workshop on Logic Based Program Synthesis and Transformation*, pages 165–182, London, UK, 2001. Springer-Verlag.

[23] C.-W. Jeon, I.-G. Kim, and J.-Y. Choi. Automatic

generation of the c# code for security protocols verified with casper/fdr. In *AINA '05: Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 507–510, Washington, DC, USA, 2005. IEEE Computer Society.

[24] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *International Conference on Software Maintenance (ICSM)*, pages 234–243. IEEE, 2007.

[25] J. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly, 2nd edition, 1992.

[26] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering*, 29(6):555–566, 2003.

[27] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1):26–35, 2007.

[28] S. Marlow. *Happy, a parser-generator for Haskell*. http://www.haskell.org/happy.

[29] S. Marlow. *A lexical analyser generator for Haskell*. http://www.haskell.org/alex.

[30] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[31] S. Peyton Jones. *A pretty printer library in Haskell*. Part of the GHC distribution at http://www.haskell.org/ghc.

[32] E. Pozniansky and A. Schuster. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice & Experience*, 19(3):327–340, 2007.

[33] D. Pozza, R. Sisto, and L. Durante. Spi2java: Automatic cryptographic protocol java code generation from spi calculus. In *AINA '04: Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, page 400, Washington, DC, USA, 2004. IEEE Computer Society.

[34] W. Pugh and N. Ayewah. Unit testing concurrent software. In *International Conference on Automated Software Engineering (ASE)*. ACM, 2007.

[35] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[36] C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming (ESOP)*.

[37] C. Sadowski and J. Yi. Sidetrack: Generalizing dynamic atomicity analysis. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2009. To appear.

[38] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[39] E. Schonberg. On-the-fly detection of access anomalies. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 285–297, 1989.

[40] V. Schuppan, M. Baur, and A. Biere. Jvm independent replay in java. *Electronic Notes in Theoretical Compututer Science*, 113:85–104, 2005.

[41] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. *SIGSOFT Software Engineering Notes*, 25(5):158–167, 2000.

[42] L. Wang and S. D. Stoller. Run-time analysis for atomicity. volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[43] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146, 2006.

[44] D. Wile. Supporting the DSL Spectrum. *Journal of Computing and Information Technology*, 9(4):263–288, 2001.

[45] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, 2005.

[46] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2005.