BEST: A Symbolic Testing Tool for Predicting Multi-threaded Program Failures

Malay K. Ganai† Nipun Arora⋆ Chao Wang† Aarti Gupta† Gogul Balakrishnan†

†NEC Labs America, Princeton, USA ⋆Columbia University, New York, USA

***Abstract*—We present a tool BEST (<u>B</u>inary instrumentation-based <u>E</u>rror-directed <u>S</u>ymbolic <u>T</u>esting) for predicting concurrency violations.[1] We automatically infer potential concurrency violations such as atomicity violations from an observed run of a multi-threaded program, and use precise modeling and constraint-based symbolic (non-enumerative) search to find feasible violating schedules in a generalization of the observed run. We specifically focus on tool scalability by devising POR-based simplification steps to reduce the formula and the search space by several orders-of-magnitude. We have successfully applied the tool to several publicly available C/C++/Java programs and found several previously known/unknown concurrency related bugs. The tool also has extensive visual support for debugging.**

I. Introduction

The growth of cheap and ubiquitous multi-processor systems and concurrent library support are making concurrent programming very attractive. However, verification of multi-threaded concurrent systems remains a daunting task especially due to complex and unexpected interactions between asynchronous threads. Concurrency bugs often arise due to atomicity violations, i.e., non-atomic execution of code regions that are intended to be atomic. Atomicity is a semantic correctness notion for concurrent programs, and is meant to capture the programmers intents [2], [3].

To expose a concurrency bug, a test case should not only provide a bug-exposing input, but also provide a bug-triggering execution interleaving. Unfortunately, testing a program for every interleaving on every test input is often practically impossible. Even for a given concrete trace, there can be exponentially many alternate interleavings (i.e., generalization) of the observed events in the trace.

To overcome the issue of exponential interleaving space, the current testing methodology [4]–[15] is focused on exploring a "meaningful" subset of thread interleaving for a given test input. CHESS tool [7], for example, restricts the set to a bounded number of pre-emptive context switches. Tools such as AtomFuzzer [8], CTrigger [9] attempt to explore low-probability interleavings using random sleep delays. Although that helps reduce the set, the unexplored set may still be very large, thereby, miss many errors. Alternatively, tools such as jPredictor [10], HAVE [11], and PENELOPE [15] predict atomicity violations in some generalization of the observed trace. However, the used generalization is restrictive (i.e., may miss errors), prediction is imprecise (i.e., may predict spurious schedules due to data abstraction), and the search method is enumerative.

To address the above mentioned issues, we propose a tool BEST (<u>B</u>inary instrumentation-based <u>E</u>rror-directed <u>S</u>ymbolic <u>T</u>esting) for predicting atomicity violation[2] in a generalization of an observed trace. Specifically, it:

- misses no errors by using a *complete* generalization,
- reduces spurious schedules using *precise modeling* of data and synchronization primitives,
- predicts atomicity violation based on *constraint-based symbolic* (non-enumerative) search,
- infers atomicity properties *automatically*, and
- dynamically *instruments unmodified binary* (of the program) for recording events.

II. BEST Framework

Our framework, as shown in Figure 1, are divided in five stages: (I) record trace events and build a concurrent trace model (CTM), (II) simplify CTM by reducing transitions and context switches, (III) infer and generate atomicity properties, (IV) find atomicity violation with property-specific symbolic analysis, and (V) provide debugging support.

Stage I. During an execution of a multi-threaded program under a test harness, we instrument a given target binary dynamically (using PIN [16]) to record various events such as synchronization and memory accesses. From these recorded events, we construct a concurrent trace model (CTM), illustrated as follows:

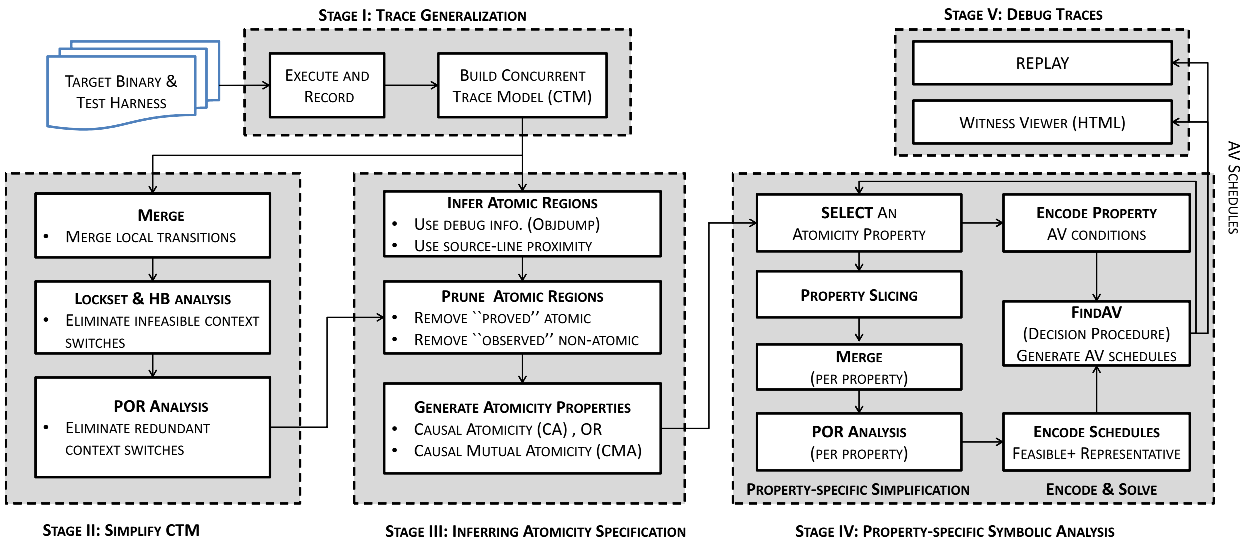
Consider a run σ of statement sequence $s_1 \cdots s_{16}$ of a concurrent program P, comprising interacting threads M_a and M_b with local variables a_i and b_i, respectively, and shared (global) data variables X, Y, Z, and synchronization variables S, L_1, L_2. This is shown in Figure 2(a) where threads are synchronized with *lock/unlock* and *wait/notify*. From the run, we obtain a CTM as shown in Figure 2(b), where a thread transition such as $(2a, true, Z := Z + 1, 3a)$ (also represented as $2a \xrightarrow{Z:=Z+1} 3a$) can be viewed as a generator of access event $RW(Z)$ corresponding to the atomic read and write access of the shared variable Z.

A CTM can be viewed as a *generalizer* of the observed trace. Alternatively, it is a *generator* for both the original run σ and all the other runs obtained by relaxing the order

[1]This tool paper highlights our scientific contributions, and summarizes the key details and experimentation as described in the paper [1].

[2]Although we skip the discussion (due to space reasons) for finding other concurrency related issues such as data races, deadlocks, and order violations, the tool has the capability to find them.

978-1-4577-1639-3/11/$26.00 © 2011 IEEE ASE 2011, Lawrence, KS, USA



Stage I: Trace Generalization
Target Binary & Test Harness
Execute and Record
Build Concurrent Trace Model (CTM)
Merge
• Merge local transitions
Lockset & HB analysis
• Eliminate infeasible context switches
POR Analysis
• Eliminate redundant context switches
Infer Atomic Regions
• Use debug info. (Objdump)
• Use source-line proximity
Prune Atomic Regions
• Remove "proved" atomic
• Remove "observed" non-atomic
Generate Atomicity Properties
• Causal Atomicity (CA), or
• Causal Mutual Atomicity (CMA)
Select An Atomicity Property
Property Slicing
Merge (per property)
POR Analysis (per property)
Encode Property AV conditions
FindAV (Decision Procedure) Generate AV schedules
Encode Schedules Feasible+ Representative
Property-specific Simplification
Encode & Solve
Stage V: Debug Traces
Replay
Witness Viewer (HTML)
AV Schedules
Stage II: Simplify CTM
Stage III: Inferring Atomicity Specification
Stage IV: Property-specific Symbolic Analysis

Fig. 1. BEST architecture

of trace events induced by non-deterministic scheduling but maintaining thread program order and fork/join semantics.

The lattice (in Figure 2(c)) represents a complete *trace generalization* of the observed trace. Each node in the lattice denotes a global control state (and a possible context switching point), shown as a pair of thread local control states. (The nodes marked with • are described in the next stage.) An edge denotes a shared event write/read/both access(es) of a global variable, labeled with $W(.)/R(.)/RZ(.)$, *lock(.)/unlock(.)*, and *wait/notify*. Note, some interleavings are infeasible due to *lock/unlock*, which we have crossed out $(\times)$ in the figure. Although the observed run σ does not show any violation, various concurrency related bugs can occur in its generalization. For instance, an alternate run (of the same statements) $\sigma' \equiv s_6 \cdot s_7 \cdot s_8 \cdot s_1 \cdot s_{12} \cdot s_2 \cdot s_{13} \cdot s_3$ can cause the following errors: *mismatched wait/notify* (s_7 before s_3, and hence, before s_5), *data races* (potential simultaneous execution of s_2 and s_{13}), *deadlock* (thread-states at s_4 and s_{14}), and an *atomicity violation* (s_{13} interrupts intended atomicity of s_2,s_3).

Trace generalization used here is less restrictive than jPredictor (where non-sequential consistent interleavings are eliminated) and PENELOPLE (where non-matched wait/notify interleavings are eliminated).

Stage II. We perform the following steps to simplify a CTM. First, we identify local (i.e., unshared) variables. Then, we *merge* a local transition (i.e., with no shared access) with the following transition in the thread program order, and rewrite the update expressions. We apply this transformation recursively. This step reduces the number of transitions by up to three times in our experiments.

We use lockset and happen-before (HB) analysis to identify the lock protected transitions and must happen-before such as fork/join, and eliminate infeasible context switches. This step gives a reduction of up to two orders-of-magnitude in the number of context switches in our experiments.

We then use a POR analysis (proposed for two or more threads [17]) to eliminate context switches corresponding to redundant interleavings (i.e., which are equivalent [18] to admitted interleavings). This step gives up to three orders-of-magnitude additional reduction in the number of context switches over lockset/HB analysis in our experiments.

The nodes marked with • in the lattice (Figure 2(c)) show the context switches allowed under POR analysis. The smaller set of context switches so obtained not only reduces the set of *necessary* thread interleavings to explore, but is also *adequate*, i.e., it includes every feasible interleaving (or its equivalent) in the CTM. From the (reduced) set of context-switches, we derive a set of independent transactions[3] (denoted as $\mathcal{I}$-transactions) such that context-switches are needed only at the begin and end of such transactions. The $\mathcal{I}$-transactions derived for threads M_a and M_b are $\{ta_1, ta_2, ta_3, ta_4\}$ and $\{tb_1, tb_2, tb_3, tb_4\}$, respectively (shown in Figure 2(b)-(c)).

Stage III. On a simplified CTM, we *infer* user-intended atomic regions (denoted as $\mathcal{E}$-transactions[4]) that may involve multiple variable accesses based on code structure. Each atomic region should satisfy the following:

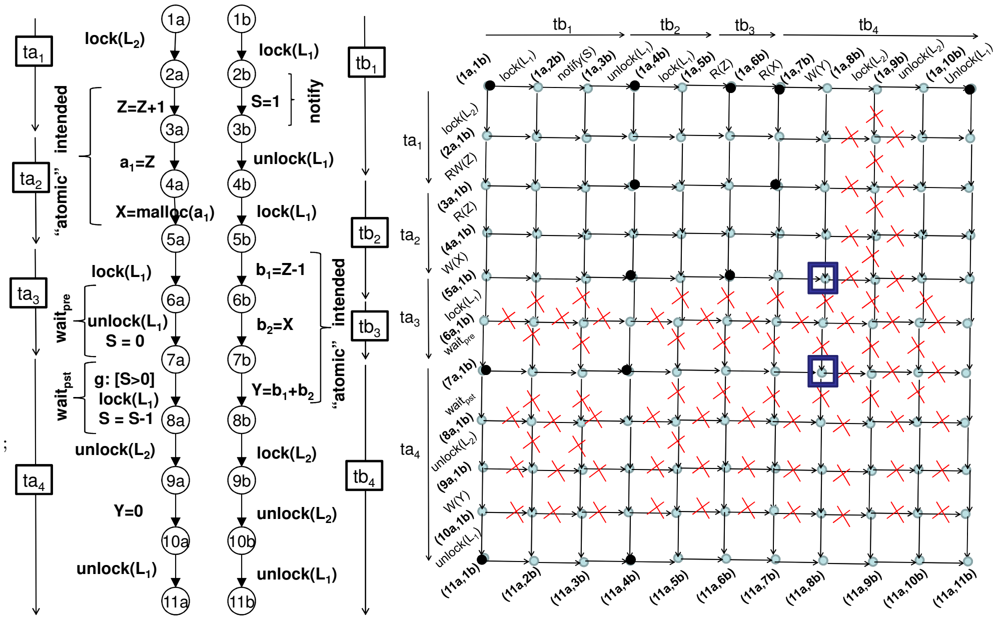
- there should be at least one shared access on a non-synchronization variable, and the first and/or last shared accesses should be on non-synchronization variable(s)
- the shared accesses should be within a procedure boundary (may include call-sites or system calls/returns)
- the source lines corresponding to shared accesses should be less than threshold distance (code statement proximity)
- no happen-before transition such as thread creation/join or wait within the region (notify is allowed)

From a given binary (assuming a debug version) we use a gnu utility such as `objdump` to obtain a mapping between processor instruction and corresponding source file and line information. We use this information to *tag* each transition with a tuple $\langle file, line\# \rangle$. Following the above guidelines, we then infer atomic regions (i.e., $\mathcal{E}$-transactions) as illustrated with

[3]An independent transaction (i.e., $\mathcal{I}$-transaction) [19] is a sequence of transitions that are atomic w.r.t to all schedules of a CTM.

[4]$\mathcal{E}$-transactions are atomicity specification that we want to validate, whereas $\mathcal{I}$-transactions are derived from the given system under test, and are used to reduce the search space of symbolic analysis.

Thread M_a	Thread M_b
s_1: $lock(L_2)$;	
s_2: a :=++Z;	
s_3: $X := malloc(a)$;	
s_4: $lock(L_1)$;	
s_5: $wait(S, L_1)$;	
	s_6: $lock(L_1)$;
	s_7: $notify(S)$;
	s_8: $unlock(L_1)$;
s_9: $unlock(L_2)$;	
s_{10}: $Y := 0$	
s_{11}: $unlock(L_1)$;	
	s_{12}: $lock(L_1)$;
	s_{13}: $Y := \&X[Z\text{-}1]$;
	s_{14}: $lock(L_2)$;
	s_{15}: $unlock(L_2)$;
	s_{16}: $unlock(L_1)$;



(a) (b) (c)

Fig. 2. (a) Executed statements in a run σ, (b) Concurrent Trace Model (CTM), (c) Lattice representing generalization of the trace σ, and a necessary and sufficient set of context switches (marked with ●) derived under partial-order reduction.

the following example. Consider an *object dump*, shown in

```
  .......
 ./atom.c:59
   pthread_mutex_lock(l2);
  .......
  8048776:       e8 c1 fd ff ff        call   804853c
 ./atom.c:61
   ++Z;
  804877b:       a1 28 9a 04 08        mov    0x8049a28,%eax
  8048780:       83 c0 01              add    $0x1,%eax
  8048783:       a3 28 9a 04 08        mov    %eax,0x8049a28
 ./atom.c:63
   X = (char *)malloc(Z);
  8048788:       a1 28 9a 04 08        mov    0x8049a28,%eax
  804878d:       89 04 24              mov    %eax,(%esp)
  8048790:       e8 97 fd ff ff        call   804852c
  8048795:       a3 2c 9a 04 08        mov    %eax,0x8049a2c
 ./atom.c:65
   pthread_mutex_lock(l1);
  80487a1:       e8 96 fd ff ff        call   804853c
  .......
```

Fig. 3. Inferring atomicity with objdump using code structure

Figure 3, for the statements $s_1 \cdots s_4$ of our running example. The transition corresponding to s_1, i.e., $(2a \to 3a)$ is assigned a tag $\langle atom.c, 61\rangle$. Similarly, the transitions corresponding to s_2, i.e., $(3a \to 4a)$ and $(4a \to 5a)$ are both assigned the tag $\langle atom.c, 63\rangle$. Using the rules for inferring atomic regions, we mark the transitions corresponding to statements s_2 and s_3 to be atomic. Similarly, we infer that the transitions corresponding to s_{13} (and s_{10}) are expected to be atomic.

We remove those regions which are observed to violate the atomicity (in the given trace). We *prune* out those which are "proved atomic" (i.e., when an inferred $\mathcal{E}$-transaction is subsumed by some derived $\mathcal{I}$-transaction). From the remaining regions, we derive atomicity properties. We use causal atomicity (CA) [20] that checks the atomicity violation of an atomic region. We also use a new notion of *causal mutual atomicity* (CMA) which checks the atomicity violation of pair-wise atomic regions corresponding to different threads with at least two conflicting transitions. Empirically, we found CMA to be more useful (in finding more violations) and simpler to check (more scalable) than CA.

Stage IV. For each derived atomicity property (CA or CMA), we first carry out property preserving (i.e., sound) slicing of the CTM (obtained in stage II). This is followed by sound simplification steps such as merging and POR analysis on the sliced CTM. These steps give additional orders-of-magnitude reduction in the number of context switches (more for a CMA than a CA property.)

We devise an algorithm FindAV for checking atomicity violation (AV) precisely. For each property, we efficiently encode the AV condition along with a symbolic set of schedules (feasible in the sliced CTM) as a quantifier-free SMT (Satisfiability Modulo Theory) formulas. We implemented a SMT-based decision procedure (in FindAV) to find a feasible schedule violating the atomicity property, i.e., *if and only if* one exists in the sliced CTM w.r.t. the property. (As checks are independent, we plan to parallelize them in future).

The highlights of our symbolic encoding are as follows: (a) it does not require an external bound on the number of context switches, (b) the worst case size of the encoded formula is quadratic in the number of observed shared access events, and (c) it leverages POR techniques to reduce *both* the formula size *and* the search space.

Stage V. For replaying a bug-triggering schedule, we instrument the binary (during runtime) to carry out an orchestrated execution using an externally provided schedule, overriding default OS scheduling. We also provide visualization of such a schedule, where each trace event is mapped to the corresponding source statements. The mapping information is derived from the objdump, where each corresponding transition is tagged with a tuple $\langle file, line\#\rangle$ (similar to stage III).

III. Implementation and Evaluation

Implementation. We use `gcc/g++/gcj` compilers to transform `C/C++/Java` programs to x86 binaries, respectively. During runtime, we instrument the application binary and dynamically loadable libraries such as `pthread` using PIN [16] to record the synchronization events such as wait/notify, lock/unlock, fork/join, sem_wait/sem_post, and heap memory accesses. We use SMT solver Yices-1.0.29 [21] in our decision procedure. We use an SMT encoding [22] (improvised over [23]) to obtain a set of feasible schedules.

For practical reasons, we do not record local accesses such as stack reads/writes. In such cases, CTM would be an abstraction of the actual program execution, and the AV schedules although feasible in CTM may be spurious, i.e., not replayable. Currently, we suppress reporting such spurious schedules, although we did not see such a case in our experimentation (due to precise modeling).[5]

Experimentation. We experimented with several multi-threaded publicly available applications [24], [25] (written in `C/C++/Java`) with 1K-33K LOC such as *aget* (1.2KLOC, `C`), *fastspy* (1.5KLOC, `C`), *finalsolution* (2KLOC, `C++`), *prozilla* (2.7KLOC, `C++`), *axel* (3.1KLOC, `C`), *bzip2smp* (6.4KLOC, `C`), *alsaplayer* (33KLOC, `C++`), and *tsp* (713, `Java`). The number of trace events ranges from a few hundreds to 34K, and number of threads ranges from 4 to 67. Most atomic regions involve multiple variable accesses. Time per check on average is around a few seconds. CMA checking generally is more robust compared to CA checking. We observed at least an order of magnitude more reduction in the number of context switches per CMA property compared to per CA property. All the found AV schedules were feasible. We found several previously known/unknown AV bugs (Bug list can be found here [26]).

The combination of various steps in stages II and IV help in reducing the context switches by a few orders of magnitude. Similarly, these steps reduce the number of transitions significantly (by an order in some cases). Overall, these reductions play a crucial role in improving the scalability of our tool.

IV. Benefits and Contributions

Although several runtime tools for predicting atomicity violation exist, namely, jPredictor [10], HAVE [11], SideTrack [12], CTrigger [9], and PENELOPE [15], our tool provides a *unique combination* of the following:

- *Generalization*: Our scope of generalization is complete. There is no restriction on admitted schedules such as nested locking, race-freedom, and matched wait/notify.
- *Precision*: Our modeling of synchronization and data path variables is precise. Our symbolic encoding captures all and only feasible schedules.
- *Scalability*: Our POR-based simplification steps reduces both the size of decision problems and the search space by orders-of-magnitude. Our prediction is based on constraint-based symbolic search, and avoids explicit enumeration of a prohibitively large set of schedules.
- *Flexibility*: Our atomicity checking algorithm handles various notions of atomicity violation.

The salient features of our tool are: effective staged simplification and reduction enabling scalable symbolic search, automatically inferring atomic regions from a concrete execution, generating atomicity violating schedules that exhibit real bugs and scope for task parallelization.

[5]In future, we plan to re-execute the binary with the non-replayable AV schedule, build a different CTM, and then apply stages I-IV.

References

[1] M. Ganai. Scalable and precise symbolic analysis for atomicity violations. In *ACM Transactions on Software Engineering Method*, 2011.

[2] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[3] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proc. of POPL*, 2006.

[4] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[5] S. Lu, J. Tucekt, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Architectural Support for Programming Languages and Operating Systems*, 2006.

[6] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Symposium on Operating Systems Principles*, 2007.

[7] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. of OSDI*, 2008.

[8] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *International Symposium on the Foundations of Software Engineering*, 2008.

[9] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[10] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A predictive runtime analysis tool for java. In *International Conference on Software Engineering*, 2008.

[11] Q. Chen, L. Wang, Z. Yang, and S. Stoller. HAVE: Detecting atomicity violations via integrated dynamic and static analysis. In *Proc. of FASE*, 2009.

[12] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: Generalizing dynamic atomicity analysis. In *Proc. of PADTAD*, 2009.

[13] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Proc. of CAV*, 2009.

[14] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *Proc. of TACAS*, 2010.

[15] F. Sorrentino, A. Farzan, and M. Parthasarathy. PENELOPE: weaving threads to expose atomicity violations. In *International Symposium on the Foundations of Software Engineering*, 2010.

[16] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[17] M. K. Ganai and S. Kundu. Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In *Proc. of SPIN Workshop*, 2009.

[18] A. Mazurkiewicz. Basic Notions of Trace Theory. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 285–363. Springer-Verlag, 1988.

[19] M. Ganai and C. Wang. Interval analysis for concurrent trace programs using transaction sequence graphs. In *Proc. of Runtime Verification*, 2010.

[20] A. Farzan and P. Madhusudan. Causal Atomicity. In *Proc. of CAV*, 2006.

[21] SRI. Yices: An SMT solver. *http://fm.csl.sri.com/yices*.

[22] M. K. Ganai. Efficient symbolic analysis for trace generalization. In *Under Submission*, 2011.

[23] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *Proc. of SPIN Workshop*, 2008.

[24] Geeknet Inc. Freshmeat. *http://freshmeat.net*.

[25] Geeknet Inc. SourceForge. *http://sourceforge.net*.

[26] M. K. Ganai. Conference Notes. *http://www.nec-labs.com/∼malay/notes.htm*.