CONTESSA: Concurrency Testing Augmented with Symbolic Analysis

Sudipta Kundu*, Malay K. Ganai, and Chao Wang

NEC Labs America, Princeton, NJ, USA

Abstract. Testing of multi-threaded programs poses enormous challenges. To improve the coverage of testing, we present a framework named CONTESSA that augments conventional testing (concrete execution) with symbolic analysis in a scalable and efficient manner to explore both thread interleaving and input data space. It is built on partial-order reduction techniques that generate verification conditions with reduced size and search space. It also provides a visual support for debugging the witness traces. We show its significance in testbeds.

1 Introduction

Concurrency testing poses a major challenge due to large *interleaving* space of concurrent programs. To expose a concurrency bug, a test case should not only provide a bug-exposing input, but also provide a bug-triggering execution interleaving. Testing a program's behavior for every interleaving on every test input is infeasible.

Dynamic model checking [1–3], for a given test input performs systematic execution of a program under different thread interleavings. Even for a fixed test input, explicit enumeration of interleavings can still be quite expensive. Although partial order reduction techniques (POR) reduce the set of interleavings to explore, the reduced set often remains prohibitively large. Some previous work use ad-hoc approaches such as perturbing program execution by injecting artificial delays after every synchronization points [4], or use randomized dynamic analysis to detect real races [5]. Although such approaches addresses scalability, often they do not provide adequate coverage.

1.1 CONTESSA Framework: Overview

To improve the coverage of testing, we present a framework named CONTESSA that augments conventional CONcurrency TESting with Symbolic Analysis in a scalable and efficient manner to explore both thread interleavings *and* the input data space, as shown in the Figure 1.

First we automatically instrument a given source code (a multi-threaded C/C++ program) for logging global access events. We then obtain an executable binary of the instrumented code. (Alternately, one can instrument the binary directly for logging the global events.) We run the binary on a given set of test cases that include monitors corresponding to reachability properties such as common program errors, data races,

* Sudipta Kundu worked on the project as an intern at NECLA. He is now at Synopsys Inc.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 127–131, 2010.

atomicity violations [6]. Corresponding to each run, we obtain a corresponding concrete execution trace. From a set of these traces, we derive a lean partition of the program called a concurrent trace program (CTP) [7]. Implicitly, such a CTP captures all linearizations of the trace events that respect the control flow of the program.

To strike a balance between coverage and scalability, we use such a derived CTP to drive our symbolic analysis, i.e., to explore various interleavings symbolically to validate a given set of reachability properties. Specifically, the search engine combines a partial-order reduction technique [8] with a token-based (asynchronous) modeling approach [9] to generate verification conditions directly without an explicit scheduler. The corresponding formula is efficiently encoded [8] to obtain reduction both in the size and the interleaving search space. Such formulas can then be solved by the state-of-the-art SMT (Satisfiability Modulo Theory) solvers (e.g. [10]) with relative ease. The witness traces corresponding to the properties can be visualized in an event trace viewer, making debugging process easier. The tool currently supports C/C++ programs on the Linux/Pthreads platforms.

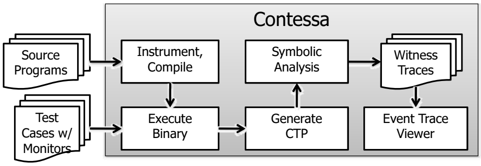
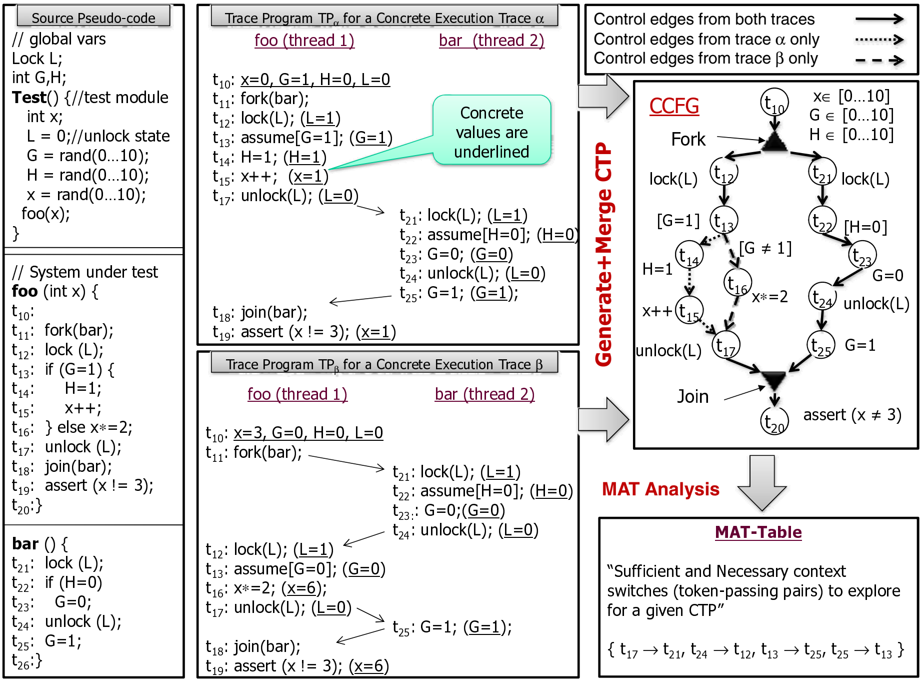


Fig. 1. Concurrent testing framework

Although one can derive the verification conditions directly from a source code (e.g. [9]), they are typically modeled imprecisely due to dynamic data elements such as pointers, linked lists, arrays and library calls. Such imprecision typically leads to spurious witnesses. To overcome the issue of spuriousness, we generate these conditions directly from CTPs which includes all the valid program traces. As a CTP is much smaller in size compared to the entire source program, it leads to manageable-sized verification conditions. Moreover, such CTPs can also be derived easily from prevalent testing infrastructures. In short, the strength of the tool is in finding "error traces" based on symbolic analysis of a set of "good traces."

2 Tool Flow

We highlight the various steps of the tool chain with an example shown in Figure 2. The example is a multi-threaded C program $\{\texttt{foo, bar}\}$ with shared variables G, H, L. The `test` harness invokes the main program `foo` with various `random` test values. The program is automatically instrumented (not shown) to log various memory accesses and synchronization events (denoted as t_i) during execution. The trace programs TP_α and TP_β correspond to two traces $\alpha = t_{10}\{t_{11}\cdots t_{17}\}\{t_{21}\cdots t_{25}\}t_{18}t_{19}$ with $x = 0, G = 1, H = 0$, and $\beta = t_{10}t_{11}\{t_{21}\cdots t_{24}\}t_{12}t_{13}t_{16}t_{17}t_{25}t_{18}t_{19}$ with $x = 3, G = 0, H = 0$, respectively of the test system. The concrete values of trace events are shown in the brackets (and underlined). Note, a trace program denotes a totally ordered events. The assertion at t_{19} denotes the correctness property, which holds for these two runs. Due to a potential race condition between t_{13} and t_{25}, the assertion may fail on a run such as $t_{10}t_{11}\{t_{21}\cdots t_{24}\}t_{12}t_{25}t_{13}\{t_{16}\cdots t_{19}\}$ with $x = 2, G = 0, H = 0$.



Source Pseudo-code

```
// global vars
Lock L;
int G,H;
Test() {//test module
  int x;
  L = 0;//unlock state
  G = rand(0...10);
  H = rand(0...10);
  x = rand(0...10);
 foo(x);
}
```

```
// System under test
foo (int x) {
$t_{10}$:
$t_{11}$:  fork(bar);
$t_{12}$:  lock (L);
$t_{13}$:  if (G=1) {
$t_{14}$:      H=1;
$t_{15}$:      x++;
$t_{16}$:  } else x*=2;
$t_{17}$:  unlock (L);
$t_{18}$:  join(bar);
$t_{19}$:  assert (x != 3);
$t_{20}$:}
```

```
bar () {
$t_{21}$:  lock (L);
$t_{22}$:  if (H=0)
$t_{23}$:     G=0;
$t_{24}$:  unlock (L);
$t_{25}$:  G=1;
$t_{26}$:}
```

Trace Program TP_α for a Concrete Execution Trace α

```
foo (thread 1)                          bar  (thread 2)

$t_{10}$: x=0, G=1, H=0, L=0
$t_{11}$: fork(bar);
$t_{12}$: lock(L); (L=1)
$t_{13}$: assume[G=1]; (G=1)
$t_{14}$: H=1; (H=1)
$t_{15}$: x++;  (x=1)
$t_{17}$: unlock(L); (L=0)
                                  $t_{21}$: lock(L); (L=1)
                                  $t_{22}$: assume[H=0]; (H=0)
                                  $t_{23}$: G=0; (G=0)
                                  $t_{24}$: unlock(L); (L=0)
                                  $t_{25}$: G=1; (G=1);
$t_{18}$: join(bar);
$t_{19}$: assert (x != 3); (x=1)
```

Concrete values are underlined

Trace Program TP_β for a Concrete Execution Trace β

```
foo (thread 1)                          bar (thread 2)

$t_{10}$: x=3, G=0, H=0, L=0
$t_{11}$: fork(bar);
                                  $t_{21}$: lock(L); (L=1)
                                  $t_{22}$: assume[H=0]; (H=0)
                                  $t_{23}$: G=0;(G=0)
                                  $t_{24}$: unlock(L); (L=0)
$t_{12}$: lock(L); (L=1)
$t_{13}$: assume[G=0]; (G=0)
$t_{16}$: x*=2; (x=6);
$t_{17}$: unlock(L); (L=0)
                                  $t_{25}$: G=1; (G=1);
$t_{18}$: join(bar);
$t_{19}$: assert (x != 3); (x=6)
```

MAT-Table

"Sufficient and Necessary context switches (token-passing pairs) to explore for a given CTP"

{ $t_{17} \rightarrow t_{21}$, $t_{24} \rightarrow t_{12}$, $t_{13} \rightarrow t_{25}$, $t_{25} \rightarrow t_{13}$ }

Fig. 2. A run of the tool

Generate CTP. From a trace, we obtain a concurrent trace program (CTP) by relaxing the order (of events) induced by non-deterministic scheduling, and maintaining only those that are induced by thread program order and fork/join semantics [7]. For the example, fork/join induces following partial-order event pairs: (t_{11}, t_{12}), (t_{11}, t_{21}), (t_{25}, t_{18}), and (t_{17}, t_{18}). We also allow data inputs to take range values as defined in the test harness. As shown, x, G and H take arbitrary values within the range $[0 \cdots 10]$.

We represent CTP_α (CTP_β), as a concurrent control flow graph $CCFG_\alpha$ ($CCFG_\beta$) (not shown separately). The control edges in a CCFG represent the partially-ordered events of the CTP. These two CCFGs can be "stitched" together by merging the respective control edges. This combined CCFG implicitly represents a "merged" CTP. The solid arrows denote the control edges common in both CTPs, whereas the dotted/dashed arrows denote the exclusive control edges from CTP_α/CTP_β, respectively.

Symbolic Analysis. All linearizations of a CTP may not correspond to actual executions of the program. For example, a linearization $\cdots t_{25}t_{13}t_{16} \cdots$ does not correspond to any executable trace (as the branch $[G \neq 1]$ will not hold). We define "feasible linearizations" of a CTP to be those that correspond to actual program executions [7]. We generate verification conditions (using the following encoding) to search within the feasible linearizations of a CTP; if an error is found, it is guaranteed to be real.

SMT-based Encoding (`mat-enc`). We use a token-passing modeling approach [9] to generate a quantifier-free first-order logic formula (ϕ). First, we create independent (uncoupled) models for each individual thread in the CTP. On such thread-models, we apply thread-local transformation and simplification [9] to reduce the thread modeling constraints (ϕ_{TM}). Second, we add token passing constraints ϕ_{TPM} between only those context-switching events as identified by MAT analysis (described next). Optionally, we add thread-specific fine grained context-bounding constraints ϕ_{CB} for more scalability, though at the cost of completeness. The formula ϕ represents the following conjunction, where ϕ_{PRP} denote the formula corresponding to the correctness property.

$$\phi = \phi_{TM} \wedge \phi_{TPM} \wedge \neg\phi_{PRP} (\ \wedge \ \phi_{CB} \ )$$

MAT Analysis. On a CTP, we use a partial-order reduction technique based on *Mutually Atomic Transactions* (MAT) [8] to identify an optimal and adequate set of context switches (or token-passing pairs) to cover the entire interleaving space of the CTP. The basic idea is as follows: a MAT is a pair of transactions (i.e., a sequence of transitions) corresponding to two threads, such that only the last transitions in the pair of transactions have conflicting shared variable accesses. An interesting observation is that there are only two different program behaviors possible by interleaving the various transitions in a MAT. As shown for the example, MAT analysis produces a necessary set of only 4 context switches, whose combination guarantees a complete thread interleaving coverage for the CTP. Such a reduced set of context switches not only reduces the interleaving search space but also the size of the formula ϕ. Due to this improvement, the tool can search a potentially a larger CTP (i.e., a larger coverage) for possible violations.

3 Evaluation

We applied CONTESSA to several case studies. In one case study [8], our goal was to check assertions that specify the functional correctness of multi-threaded programs. We used random test vectors to generate CTPs of trace depths of 400. Note, though each test vector may produce a distinct trace (CCFG), typically it differs from the rest only in a few control edges. By stitching these CCFGs, as mentioned earlier, we obtain a compact CTP. In these CTPs, the number of threads was 2 to 3, the number of shared accesses was between 4 and 200 per thread. The total number of possible context-switches was between 50-800K. After MAT analysis, the number of context-switches was reduced to 14-2500. This reduction directly translated in the size reduction of verification conditions from the range 32K-48M to 26K-3.3M. On a few of these CTPs, we found assertion violations in less than a minute.

In another case study, we applied our tool to obtain CTPs from execution traces generated by Java PathFinder [11]. The test programs include publicly available multi-threaded Java benchmarks such as *hdec*, *Daisy*, and *Tsp*. The trace lengths range from 200 to 45K, and the number of threads ranges from 3 to 21. In these generated CTPs, the number of lock/unlock events ranges from 4 to 1K, and the number of wait/notify events ranges from 0 to 41. Our symbolic analysis algorithm were able to find data races in a few of these CTPs within one minute, producing corresponding witness traces. Thus, our tool can effectively be applied as a plug-in module in prevalent testing infrastructures to improve the test coverage of concurrent programs.

Overall, we believe that the tool is a promising compromise between scalability of testing and coverage of symbolic static analysis.

References

1. Godefroid, P.: Software Model Checking: The Verisoft approach. In: FMSD (2005)
2. Musuvathi, M., Quadeer, S.: CHESS: Systematic stress testing of concurrent software. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 15–16. Springer, Heidelberg (2007)
3. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah (2008)
4. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for Testing Multi-threaded Java Programs. In: Concurrency and Computation: Practice and Experience (2003)
5. Sen, K.: Race directed random testing of concurrent programs. In: PLDI (2008)
6. Farzan, A., Madhusudan, P.: Causal Atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
7. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: ESEC-FSE (2009)
8. Ganai, M.K., Kundu, S.: Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In: Proc. of SPIN Workshop (2009)
9. Ganai, M.K., Gupta, A.: Efficient modeling of concurrent systems in bmc. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 114–133. Springer, Heidelberg (2008)
10. SRI. Yices: An SMT solver, `http://fm.csl.sri.com/yices`
11. JPF, `http://babelfish.arc.nasa.gov/trac/jpf`