Concolic Testing of Concurrent Programs

Azadeh Farzan[1], Andreas Holzer[2], Niloofar Razavi[1], and Helmut Veith[2]

[1] University of Toronto, CAN
[2] Vienna University of Technology, AT

Abstract. We survey our research on concolic testing of concurrent programs. Based on *conc*rete and symb*olic* executions of a *con*current program, our testing techniques derive inputs and schedules such that the execution space of the program under investigation is systematically explored. We will focus on (con)^{2}colic testing which introduces *interference scenarios*. Interference scenarios capture the flow of data among different threads and enable a unified representation of path and interference constraints.

1 Overview

Bounded-Interference-based Sequentialization. In [1], we proposed an approach that is based on a program transformation technique that takes a concurrent program P as an input and generates a sequential program that simulates a subset of behaviors of P. It is then possible to use an available sequential testing tool to test the resulting sequential program. We introduce a new interleaving selection technique, called *bounded-interference*, which is based on the idea of limiting the degree of *interference* from other threads. An interference occurs when a thread reads a value that is generated by another thread. Our sequentialization technique encodes all interleavings of P with a certain interference degree k in the resulting sequential program $\widehat{P}_k$. All input variables of P are retained as input variables of $\widehat{P}_k$, and new input variables are introduced that encode interleaving choices. It is then possible to use an available sequential testing tool (in our case, PEX) to test the resulting sequential program $\widehat{P}_k$, which through standard systematic input generation for $\widehat{P}_k$, performs both input generation and interleaving exploration for P. The transformation is sound in the sense that any bug discovered by a sequential testing tool in the sequential program is a bug in the original concurrent program yet it lacks completeness.

(Con)^{2}colic Testing. To resolve the lack of completeness for bounded programs in [1], we introduced (con)^{2}colic testing [2], an approach that systematically explores both input and interleaving spaces of concurrent programs. (Con)^{2}colic testing can provide meaningful coverage guarantees during and after the testing process. (Con)^{2}colic testing can be viewed as a generalization of sequential concolic (*conc*rete and symb*olic*) testing [3] to concurrent programs that aims to achieve maximal code coverage for the programs. Like [1], (con)^{2}colic testing also exploits interferences among threads: The central objects in (con)^{2}colic testing are *interference scenarios*. An interference scenario represents a set of interferences among threads. Conceptually, interference scenarios describe the prefix of a concurrent program run such that all program runs with the same interference scenario follow the same control flow during execution of that prefix. By systematically enumerating interference scenarios, (con)^{2}colic testing explores the input and scheduling space of a concurrent program to generate tests (i.e., input values and a schedule) that cover a previously uncovered part of the program. We first enumerate all feasible interference scenarios that involve no interference what amounts to enumerating purely thread-local behaviors. After we have completed that, we continue with all feasible interference scenarios that involve one interference, then two interferences, then three, and so forth. Not all

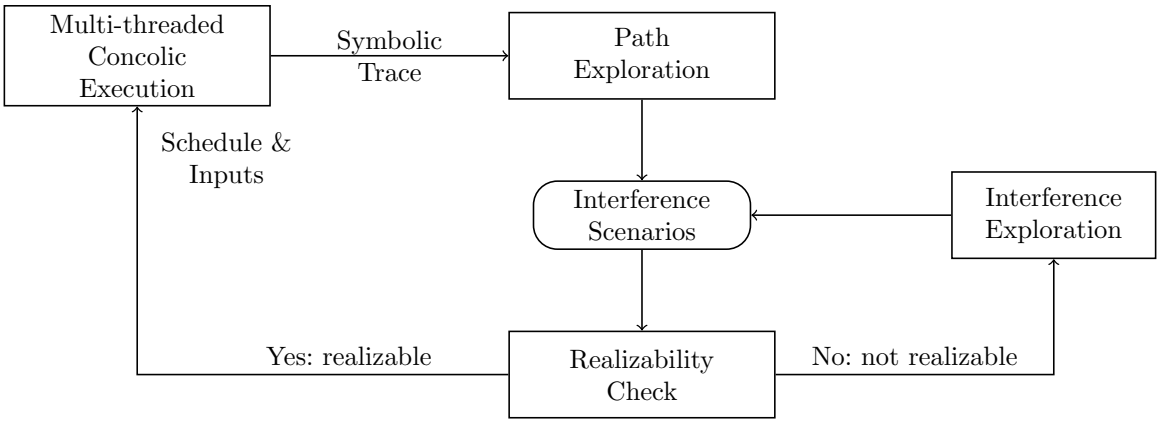

Fig. 1. Overview of (Con)^{2}colic Testing.

interference scenarios that we consider for enumeration are feasible. The interference scenarios involving $k+1$ interferences are either observed during a concrete program execution or are constructed from an extension of an infeasible interference scenario with an interference from a feasible interference scenario (both having k or less interferences).

Figure 1 shows the four main components of our (con)^{2}colic testing framework: **(1)** A *concolic execution engine* executes the concurrent program according to a given input vector and schedule. The program is instrumented such that, during the execution, all important events are recorded. This information is used to generate further interference scenarios. **(2)** A *path exploration component* decides what *new* scenario to try next, aiming at covering previously uncovered parts of the program. **(3)** A *realizability checker* checks for the realizability of the interference scenario provided by the path exploration component. Based on this interference scenario it extracts two constraint systems (one for the input values and one for the schedule) and checks for the satisfiability of them. If both are satisfiable, then the generated input vector and the schedule are used in the next round of concolic execution. **(4)** An *interference exploration component* extends unrealizable interference scenarios by introducing new interferences. (Con)^{2}colic testing can be instantiated with different search strategies to explore the interference scenario space.

2 Related Work

White-box testing concurrent programs has been a very active area of research in recent years. To alleviate the interleaving explosion problem that is inherent in the analysis of concurrent programs a wide range of *heuristic-based* techniques have been developed. Most of these techniques [4–7] do not provide meaningful coverage guarantees, i.e., a precise notion of what tests cover. Other such techniques [8] provide coverage guarantees only over the space of interleavings by fixing the input values during the testing process.

Sequentialization techniques [9] translate a concurrent program to a sequential program that has the same behavior (up to a certain context bound), and then perform a *complete static symbolic* exploration of both input and interleaving spaces of the sequential program for the property of interest. Sequentialization of concurrent executions based on linear interfaces [10] bounds the number of context switches that they consider during state-space exploration. In contrast, (con)^{2}colic testing does not put restrictions on the number of context switches that occur when computing an input vector and an interleaving that realize an interference scenario. In fact, an interference scenario with only one interference might require a huge

number of context switches due to synchronization. Note that interferences do not refer to locks, i.e., they only refer to read and write accesses of data variables. Locks are handled in a separate constraint system that expresses temporal constraints on the events of an execution. (Con)^{2}colic testing instead puts a bound on the number of interferences. On the other hand, sequentialization based on linear interfaces [10] does not put any restriction on the number of interferences that may happen during an execution that covers a linear interface. Both techniques therefore represent different strategies in exploring the state space of a concurrent program.

Symbolic PathFinder is a test generator that combines symbolic execution, model checking, and constraint solving[3]. It uses on-the-fly partial order reduction to limit the interleavings that have to be considered during state-space exploration. In contrast, (con)^{2}colic testing does not guide its exploration on interleavings but rather enumerates interference scenarios. An interference scenario imposes constraints on the input space and interleavings such that the scenario represents the set of combinations of input vectors and interleavings that trigger executions that cause exactly the same flow of data and control as specified in the interference scenario. For concrete execution, (con)^{2}colic testing then obtains one combination of input vector and interleaving from this set by solving the corresponding constraint systems.

3 Experiments

To evaluate (con)^{2}colic testing we have implemented the tool ConCrest[4] [2]. It supports multi-threaded C programs and uses a search strategy that targets assertion violations and explores interference scenarios according to the number of interferences in an ascending order. This exploration strategy is complete modulo the explored interference bound and produces minimal error traces (wrt. the number of interferences). We ran our experiments on a dual-core 64-bit Linux machine with 3.2GHz and 16GB RAM.

Benchmarks. `bluetooth` is a simplified version of the Bluetooth driver from [11]. `sor` is from Java Grande multi-threaded benchmarks (which we translated to C). `ctrace-a` and `ctrace-b` are two test drivers for the `ctrace` library. `apache-a` and `apache-b` are test drivers for APACHE FTP server from BugBench [12]. `splay` and `rbTree` are test drivers for a C library implementing several types of trees. `aget` is a multi-threaded download accelerator. `pfscan` is a multi-threaded file scanning program. Finally, `art` is an example designed by us to evaluate the scalability of our approach when the number of threads increase. It has the property that there is a new assertion in it every time we increase the number of threads by one.

Experimental Results. In our experiments we set $k_{max} = 100$ (at most 100 interferences) and a timeout of 2 hours. The results are presented in Table 1. We learned the following important facts: **(i)** ConCrest is effective at finding bugs. All the known bugs were discovered. **(ii)** All bugs discovered by ConCrest in benchmarks were the result of a branch which would not be covered sequentially. **(iii)** All bugs were discovered under a relatively small number of interferences (max. 4). **(iv)** On average, a substantial number of branches were not sequentially coverable and were only covered after interferences were introduced, e.g., for `rbTree` which has fixed input, branch coverage increases from 67 to 95 (maximum number of coverable branches). In the lack of a bug found, reaching this maximum provides guarantees to the tester that, e.g., no assertions in the code can be violated. **(v)** We set the maximum number of interferences to be 100, but the actual bound explored by ConCrest is much smaller. This is because in most cases (with the exception of

[3] `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`, accessed last on 2014-06-05.
[4] `http://forsyte.at/software/concrest/`

Benchmark	♯Ts	♯Is	♯Branches (total)	♯Branches k=0/1/2/3/4/...	Max k reached (reason)	♯Branches k=0 → Max k	Bug found (k)	♯ISC (total)	t [s] (total)
`bluetooth`	3	2	24	14/8/2	2 (FC)	14→24	yes(2)	282	0.5
`sor`	3	-	48	37/8/0/0/3	4 (FC)	37→48	yes(3)	145	0.6
`ctrace-a`	3	-	94	54/3	5 (MC)	54→57	yes(1)	28	0.7
`apache-a`	3	3	72	41/0/1	11 (MC)	41→42	yes(2)	392	1.0
`splay`	3	-	112	46/14/4	15 (MC)	46→64	no	3501	6.2
`apache-b`	3	3	48	35/3	11 (MC)	35→38	yes(1)	22150	15.4
`aget`	3	-	88	56/0/1	21 (MC)	56→57	yes(2)	23197	170.4
`rbTree`	3	-	146	67/22/4/2	24 (MC)	67→95	no	77037	296.3
`pfscan`	3	2	130	92/0/0/0/1	4 (TO)	92→93	yes(4)	3012548	7200.0
`ctrace-b`	3	-	128	75/5	2 (TO)	76→81	yes(1)	315639	7200.1
`art2`	3	2	8	7/1	1 (FC)	7→8	yes(1)	80	0.3
`art3`	4	3	12	10/1/1	2 (FC)	10→12	yes(2)	17942	21.8
`art4`	5	4	16	13/1/1/1	3 (FC)	13→16	yes(3)	2842066	197.1
`art5`	6	5	20	16/1/1/1/1	4 (FC)	16→20	yes(4)	10851573	741.1

♯**Ts**: number of threads. ♯**Is**: number of inputs. ♯**Branches**: number of *static* branches, i.e. number of basic code blocks. k: number of interferences. **FC**: all branches are covered. **MC**: all possible interference scenario candidates are explored. **TO**: time out. ♯**ISC**: number of explored interference scenario candidates. "14/8/2" means 14 branches covered at $k = 0$, 8 (new) branches at $k = 1$, and 2 (new) branches at $k = 2$. $14 \rightarrow 24$ indicates the difference between the number of branches covered sequentially (14) and the total number of branches covered (24).

Table 1. Experimental Results.

2 timeout cases), we either achieved full branch coverage or explored all possible ISCs (i.e. no more branches are coverable). **(vi)** Our approach scales well as the number of threads increase; see `art`.

There are cases where maximum branch coverage is achieved, but the number does not coincide with the total number of static branches. These are due to (sanity-check type) assertions in the code which were never meant to be violated.

Acknowledgements

This work was supported by the Canadian NSERC Discovery Grant, the Vienna Science and Technology Fund (WWTF) grant PROSEED, and the Austrian National Research Network projects S11403, S11405, S11402-N23 (RiSE) of the Austrian Science Fund (FWF).

References

1. Razavi, N., Farzan, A., Holzer, A.: Bounded-Interference Sequentialization for Testing Concurrent Programs. In: ISoLA. (2012) 372–387
2. Farzan, A., Holzer, A., Razavi, N., Veith, H.: Con2colic testing. In: FSE. (2013) 37–47
3. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: PLDI. (2005) 213–223
4. Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic Predictive Analysis for Concurrent Programs. In: FM. (2009) 256–272
5. Sorrentino, F., Farzan, A., Madhusudan, P.: PENELOPE: Weaving Threads to Expose Atomicity Violations. In: FSE. (2010) 37–46

6. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: CAV. (2006) 419–423
7. Razavi, N., Ivancic, F., Kahlon, V., Gupta, A.: Concurrent Test Generation Using Concolic Multi-trace Analysis. In: APLAS. (2012) 239–255
8. Musuvathi, M., Qadeer, S., Ball, T.: CHESS: A Systematic Testing Tool for Concurrent Software (2007)
9. Lal, A., Reps, T.: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. Formal Methods in System Design **35** (2009) 73–97
10. La Torre, S., Madhusudan, P., Parlato, G.: Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In: CAV. (2010) 629–644
11. Qadeer, S., Wu, D.: KISS: Keep It Simple and Sequential. In: PLDI. (2004) 14–24
12. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: BugBench: Benchmarks for Evaluating Bug Detection Tools. In: Workshop on the Evaluation of Software Defect Detection Tools. (2005)