Concolic Testing of Concurrent Programs*

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

[1] Computer Science Department
University of Toronto
40 St. George Street
Toronto, Ontario M5S 2E4, Canada

[2] Vienna University of Technology
Institut für Informationssysteme 184/4
Favoritenstraße 9-11
A-1040 Vienna, Austria

Abstract: We describe (con)^{2}colic testing — a systematic testing approach for concurrent software. Based on *conc*rete and symb*olic* executions of a *con*current program, (con)^{2}colic testing derives inputs and schedules such that the execution space of the program under investigation is systematically explored. We introduce *interference scenarios* as key concept in (con)^{2}colic testing. Interference scenarios capture the flow of data among different threads and enable a unified representation of path and interference constraints.

Overview

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 [WKGG09, SFM10, SA06, RIKG12] do not provide meaningful coverage guarantees, i.e., a precise notion of what tests cover. Other such techniques [MQB07] provide coverage guarantees only over the space of interleavings by fixing the input values during the testing process. *Sequentialization* techniques [LR09] 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. However, the sequential programs generated are not appropriate models for dynamic test generation due to the nondeterminism they involve. Recently, dynamic test generation was applied to sequentialized programs [RFH12]. Yet, this approach lacks completeness.

We propose (con)^{2}colic testing, a new and systematic approach for the systematic exploration of 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 [GKS05] to concurrent programs that aims to achieve maximal code coverage for the programs. (Con)^{2}colic testing exploits *interferences* among threads. An interference occurs when a thread reads a value that is generated by another thread. We introduce the new concept of *interference scenario* as a representation of 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

*This is a summary of [FHRV13]. 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 S11403-N23 (RiSE) of the Austrian Science Fund (FWF).

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.

Our (con)^{2}colic testing framework has four main components: **(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.

To evaluate our approach we have implemented the tool ConCrest[1] [FHRV13]. 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).

References

[FHRV13] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *Proc. ESEC/SIGSOFT FSE*, pages 37–47, 2013.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.

[LR09] A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. *Formal Methods in System Design*, 35:73–97, 2009.

[MQB07] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software, 2007.

[RFH12] N. Razavi, A. Farzan, and A. Holzer. Bounded-Interference Sequentialization for Testing Concurrent Programs. In *ISoLA*, pages 372–387, 2012.

[RIKG12] N. Razavi, F. Ivancic, V. Kahlon, and A. Gupta. Concurrent Test Generation Using Concolic Multi-trace Analysis. In *Proc. APLAS*, pages 239–255. Springer, 2012.

[SA06] K. Sen and G. Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[SFM10] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proc. FSE*, pages 37–46. ACM, 2010.

[WKGG09] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta. Symbolic Predictive Analysis for Concurrent Programs. In *Proc. FM*, pages 256–272. Springer, 2009.

[1] `http://forsyte.at/software/concrest/`