RESEARCH ARTICLE OPEN ACCESS

Identification of classification criteria for testing Concurrent Software Systems

Manju Susan Thomas, MPhil
**(Assistant Professor, Department of Computer Science and Applications, SAFI Institute of Advanced Study, Vazhayoor, Kerala, India)*
Email: manjuthomasmail@gmail.com)

ABSTRACT
Many recent software systems are composed of multiple execution flows that run simultaneously, spanning from applications designed to exploit the power of recent multi-core architectures to distributed systems consisting of multiple components deployed on different physical systems. We collectively refer to such systems as concurrent systems. Concurrent systems are difficult to test, since the problems that derive from their concurrent nature depend on the interleavings of the actions performed by the individual execution flows. Testing techniques that target this problem must take into account the concurrency aspects of the systems. The increasingly rapid spread of parallel and distributed architectures led to a deluge of concurrent software systems, and the explosion of testing techniques for such systems in the last decade. The current lack of a comprehensive classification, analysis and comparison of the more testing techniques for concurrent systems limits the understanding of the strengths and weaknesses of each approach and hampers the future advancements in the field. This study provides a framework to capture the key features of the available techniques to test concurrent software systems, identifies a set of classification criteria to review and compare the available techniques, and discusses in details their strengths and weaknesses, leading to a thorough assessment of the field and paving the road for future progresses.
Keywords: Classification Criteria, Testing, Concurrent Systems, Parallel Systems, Distributed Systems

---

Date of Submission: 25-08-2017 Date of acceptance: 09-09-2017

I. INTRODUCTION

Concurrent software systems are composed of multiple execution flows that execute simultaneously, and the need to synchronize the execution flows leads to new problems and introduces new design and verification challenges. The behavior of concurrent systems depends not only on the sequence of actions executed within each individual flow, but also on the interleavings of the actions in the different execution flows. Wrong interleavings may lead to concurrency faults regardless of the correctness of the computation of each execution flow. The problem of developing reliable concurrent systems has attracted a lot of interest in the software engineering community, and has led to several solutions for designing, implementing, and refactoring, modeling, verifying and validating concurrent software systems.

Concurrency faults are intrinsically non-deterministic, since they occur only in the presence of specific interleavings, and the interleavings depend on execution conditions that are not under the direct control of the program. The testing techniques that address the problem of efficiently exploring the space of the interleavings consider one or more of the following activities: (i) generating test cases, which are sequences of operations that stimulate the system; (ii) selecting a subset of interleavings for the execution flows; (iii) executing the system with the selected test cases and interleavings and validating the results.

Although the problem of testing concurrent systems has attracted the attention of the research community since the late seventies, and has grown considerably in the last decade, to the best of our knowledge a precise survey and classification of the progresses and the results in the field is still missing.

In this paper, we provide a comprehensive survey of the state-of-the-art in testing concurrent software systems. We studied the recent literature by systematically browsing the main publishers and scientific search engines, and we traced back the results to the seminal work of the last forty years.

We present a general framework that captures the different aspects of the problem of testing concurrent software systems and that we use to identify a set of classification criteria that drive the survey of the different approaches. The survey classifies and compares the state-of-the-art techniques, discusses their advantages and limitations, and indicates open problems and possible research directions in the area of testing concurrent software systems.

1. CONCURRENT SOFTWARE SYSTEMS

In this section, we define the scope of our analysis and introduce the terminology that we adopt in this paper. To do so, we define a conceptual framework that captures the main elements of the different approaches to test concurrent software systems. In the remainder of the paper, we use the framework to structure our survey.

1.1 CONCURRENT SYSTEMS

A system is concurrent if it includes a number of execution flows1 that can progress simultaneously, and that interact with each other. This definition encompasses both flows that execute in overlapping time frames, like concurrent programs executed on multi-core, multi-processor parallel and multi-node distributed architectures, and flows that execute only in non-overlapping frames, like concurrent programs executed on single-core architectures. Depending on the specific architecture and programming paradigm, execution flows can be concretely implemented as processes on different physical machines, processes within the same machine or threads within the same process, as common in modern programming languages such as C++, Java, C# and Erlang.

We distinguish two classes of concurrent systems based on the mechanism they adopt to enable the interaction between execution flows, shared memory and message passing systems. In shared memory systems, execution flows interact by accessing a common memory. In message passing systems, execution flows interact by exchanging messages. Message passing can be used either by execution flows hosted on the same physical node or on different physical nodes (distributed systems). Conversely, shared memory mechanisms are only possible when the execution flows are located on the same node (as in multi-threaded systems).

We model a shared memory as a repository of one or more data items. A data item has an associated value and type. The type of a data item determines the set of values it is allowed to assume. We model the interaction of an execution flow f with the repository using two primitive operations: write operations wx(v), meaning that f updates the value of the data item x to v, and read operations rx(v), meaning that f reads the value v of x. Operations are composed of one or more instructions. Instructions are atomic, meaning that their execution cannot be interleaved with other instructions, while operations are in general not atomic. This model captures both operations on simple data, like primitive variables in C, and operations on complex data structures like Java objects, where types are classes, data items are objects and operations are methods that can operate only on some of the fields of the objects.

We model message passing systems using two primitive operations: send operations sf (m) that send a message m to the execution flow f, and receive operations rf (m) that receive a message m from the execution flow f. Message passing can be either synchronous or asynchronous. An execution flow f that sends a synchronous message sf' (m) to an execution flow f' must wait for f' to receive the message m before continuing, while an execution flow f that sends an asynchronous message sf' (m) to an execution flow f' can progress immediately without waiting for m to be received by f'.

The message passing paradigm can be mapped to the shared memory paradigm by modeling a send primitive as a write operation on a shared queue and a receive primitive as a read operation on the same shared queue. Thus, without loss of generality, we refer to shared memory systems in most of the definitions and examples presented in this survey.

1.2 INTERLEAVING OF EXECUTION FLOWS

The behavior of a concurrent system depends not only on the input parameters and the sequences of instructions of the individual flows, but also on the interleaving of instructions from the different execution flows that comprise the system.

We introduce the main concepts of concurrency under the assumption of a sequentially consistent model. This model guarantees that all the execution flows in a concurrent system observe the same order of instructions, and that this order preserves the order of instructions defined in the individual execution flows. We discuss the implications of relaxing this assumption at the end of this subsection, and in the survey we consider approaches regardless of this assumption. Under the assumption of sequential consistency, we can model the interleaving of instructions of multiple execution flows in a concrete program execution with a history, which is an ordered sequence of instructions of the different execution flows.

In a shared memory system, histories include sequences of invocations of read and write operations on data items. Since in general the operations on shared data items are not atomic, we model the invocation and the termination of an operation op as two distinct instructions. The execution of an operation o' overlaps the execution

of another operation o if the invocation of o' occurs between the invocation and the termination of o. In a message passing system, histories include sequence of atomic send and receive operations.

II. TESTING CONCURRENT SYSTEMS

In this paper we focus on software testing techniques that target concurrency faults, which are faults caused by unexpected interleavings of instructions of otherwise correct execution flows. Concurrency faults can be extremely hard to reveal and reproduce, since they manifest only in the presence of specific interleavings that may be rarely executed. To expose concurrency faults, testing techniques for concurrent systems need to sample not only a potentially infinite input space, but also the space of possible interleavings, which can grow exponentially with the number of execution flows and the number of instructions that comprise the flows.

The many approaches for testing concurrent systems that have been proposed so far address different aspects of the problem. Our detailed analysis of the literature led to a simple conceptual framework that captures the different aspects of the problem and relates the many approaches for testing concurrent systems. Figure 1 presents the conceptual framework that we use to provide a comprehensive view of the problem and to organize this survey.

Approaches for testing concurrent systems deal with specific types of target systems and address one or more of the three main aspects of the problem visualized with rectangles in Figure 1: generating test cases, selecting interleavings and comparing the results with oracles. Generating test cases amounts to sample the program input space and produce a finite set of test cases to exercise the target system. Selecting interleavings amounts to augment the test cases with different interleavings of the execution flows to exercise the operations that process the same input data in different order. Comparing the results with test oracles amounts to checking the behavior of the target system with respect to some oracles. The approaches that we found in the literature focus on generating test cases or selecting interleavings, sometimes dealing or comparing with oracles as well.

Figure 1 presents a conceptual framework for the testing techniques, but does not prescribe a specific process. Some approaches may first generate a set of test cases and a set of relevant interleavings and then compare the execution results with oracles, while other approaches may alternate the selection of interleavings and the comparison with oracle by executing each interleaving as soon as identified.

The approaches for generating test cases sample the input space to produce a finite set of test cases by considering the target system. They optionally also consider a target property of interleaving, a system model that provides additional information about the target system, or both.

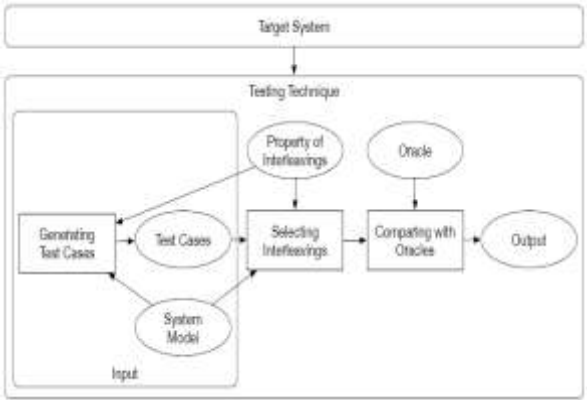

Fig. 1: A general framework for testing concurrent software systems

The approaches for selecting interleavings identify a subset of relevant interleavings to be executed, and target either the interleaving space as a whole or some specific propertie of interleaving. The techniques that target the interleaving space as a whole, hereafter space exploration techniques, explore the space of interleavings randomly, exhaustively or driven by some coverage criteria or heuristics. Two relevant classes of space exploration techniques are stress testing and bounded search techniques.

III. TRENDS IN RESEARCH ON TESTING CONCURRENT SYSTEMS

In this section we present an analysis of the research on testing concurrent systems conducted in the last fifteen years referring to the seminal work of the last forty years. Concurrency has been investigated since the early sixties with pioneer work on models for concurrent systems, like the research of Karl Adam Petri the inspiring work on process algebras of Tony Hoare [2] and Robin Milner [3].

In the seventies, with the emergence of distributed architectures, the focus of the research extended towards the analysis and verification of distributed systems with the Lipton's influential work on the theory of reduction [4] and Lamport's seminal work on distributed systems [1].

The nineties have seen the introduction of the term testing concurrent systems with continuity in the literature [5], [6], [7], and the appearing of analysis techniques that are at the core of many popular approaches for testing concurrent systems [8], [9], [10], [11], [12].

The research on testing concurrent systems has emerged overbearing in the last fifteen years fostered by the rapid spread of multi-core technologies, distributed, Web and mobile architectures and novel concurrent paradigms. Our survey indicates that most of the concurrent software testing techniques developed in the last fifteen years target shared memory systems, and only few cope with (distributed) message passing systems, which are addressed mainly by runtime monitoring and model based verification approaches.

To provide a comprehensive survey of the emerging trends in testing concurrent software systems, we systematically review the literature from 2000 to 2015: (i) we searched the online repositories of the main scientific publishers, including IEEE Explore, ACM Digital Library, Springer Online Library and Elsevier Online Library, and more generally the Web through the popular online search engines such as Google Scholar and Microsoft Academic Search.

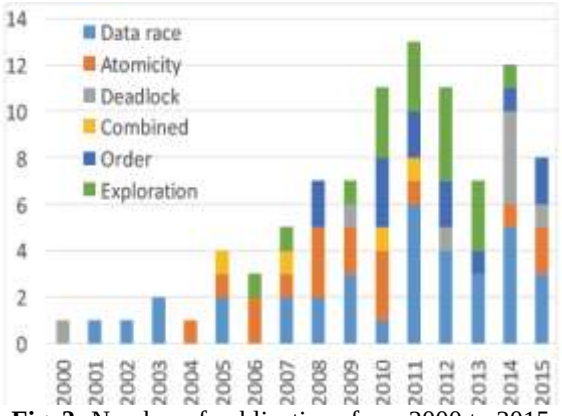

Fig. 2: Number of publications from 2000 to 2015 that witness novel research contributions and address different concurrency properties

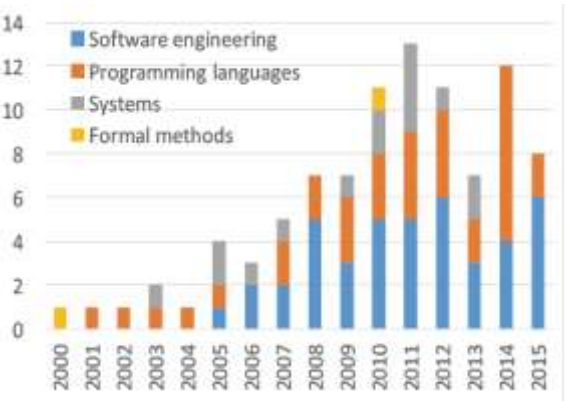

Fig. 3: Number of publications from 2000 to 2015 that witness novel research contributions in different research communities

IV. CONCLUSION

The current research trends are towards predictive property based techniques and violations of expected order invariants rather then low level memory access conflicts such as data races.

The research of the last decade has produced several efficient and effective testing techniques for concurrent systems that open promising directions for future investigations:

i) Most testing techniques for concurrent systems target the selection of relevant interleavings, and few techniques focus on test case generation. Exploiting the synergy between these two aspects remains an open research topic.

(ii) The vast majority of testing approaches target shared memory systems. Validation and verification of distributed message passing systems has exploited mostly static analysis and model checking approaches, leaving the important area of testing message passing systems open for future research.

(iii) The last decade has seen a bloom of new programming paradigms for concurrent software systems, which enforce patterns of interactions among execution flows that prevent the occurrence of some kinds of concurrency faults such as data races and deadlocks. The new programming paradigms shift the testing problem from low level memory access conflicts to high level order violations, and open the opportunity of devising new testing approaches that exploit the semantics of modern programming paradigms.

ACKNOWLEDGEMENTS

Author acknowledge the support from peer group at SIAS, Kerala

REFERENCES

[1]. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," IEEE Transactions on Computers, vol. C-28, no. 9, pp. 690–691, Sept 1979.

[2]. C. A. R. Hoare, "Communicating sequential processes," Communications of the ACM, vol. 21, no. 8, pp. 666–677, 1978.

[3]. R. Milner, A Calculus of Communicating Systems. Springer, 1982.

[4]. R. J. Lipton, "Reduction: A method of proving properties of parallel programs," Communications of the ACM, vol. 18, no. 12,pp. 717–721, 1975.

[5]. S. Morasca and M. Pezz`e, "Using high-level petri nets for testing concurrent and real-time systems," Real-time systems: theory and applications, vol. 132, 1990.

[6]. R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," IEEE Transactions on Software Engineering, vol. 18, no. 3, pp. 206–215, 1992.

[7]. R. H. Carver and K.-C. Tai, "Use of sequencing constraints for specification-based testing of concurrent programs," IEEE Transactions on Software Engineering, vol. 24, no. 6, pp. 471–490, 1998.
[8]. M. P. Herlihy and J. M.Wing, "Linearizability: A correctness condition for concurrent objects," ACM Transactions on Programming Languages and Systems, vol. 12, no. 3, pp. 463–492, 1990.
[9]. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," ACM Transactions on Computer Systems, vol. 15, no. 4,pp. 391–411, 1997.
[10]. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Transactions on Programming Languages and Systems, vol. 8, no. 2, pp. 244–263, 1986.
[11]. P. Godefroid, "Model checking for programming languages using verisoft," in Proceedings of the Symposium on Principles of Programming Languages, ser. POPL '97. ACM, 1997, pp. 174–186.
[12]. G. J. Holzmann, "The model checker spin," IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279–295, 1997.

International Journal of Engineering Research and Applications (IJERA) is **UGC approved** Journal with Sl. No. 4525, Journal no. 47088. Indexed in Cross Ref, Index Copernicus (ICV 80.82), NASA, Ads, Researcher Id Thomson Reuters, DOAJ.

Manju Susan Thomas. "Identification of classification criteria for testing Concurrent Software Systems ." International Journal of Engineering Research and Applications (IJERA) , vol. 7, no. 9, 2017, pp. 53–57.