

Non-deterministic Testing of Concurrent Programs

Jeff Lei

Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, TX 76019
ylei@cse.uta.edu

Abstract—Non-deterministic testing is the most widely used approach for testing concurrent programs in practice. However, it is typically conducted in an ad-hoc manner. In this paper, we describe a coverage-based, adaptive framework to improve the efficiency of non-deterministic testing.

I. INTRODUCTION

Concurrent programs behave differently than sequential programs. Multiple executions of a concurrent program with a fixed input may exercise different sequences of synchronization events (or SYN-sequences) and produce different results. This non-deterministic behavior makes testing concurrent programs notoriously difficult.

One approach to dealing with such difficulty is non-deterministic testing, which executes a program with a fixed input many times in hope that faults will be exposed by one of these executions. Due to its simplicity, non-deterministic testing is *the* most widely used approach in practice. However, it is typically conducted in an *ad hoc* manner. Most research has focused on how to efficiently insert noise makers (e.g. random delays) into selected program locations so that different SYN-sequences are likely to be exercised by repeated executions and thus increase the chance of finding faults [1] [2] [3]. Several problems have not been addressed yet: (1) How to measure test progress? (2) When to stop the testing process? (3) It is possible that after a certain point, repeated executions will not exercise new SYN-sequences. How can further progress be made in this situation?

In this paper, we describe a coverage-based, adaptive framework to address the above problems. The main challenge stems from the fact that coverage criteria are usually defined with respect to a graph model. To accurately measure test progress against these criteria requires constructing the graph model, which is often computationally expensive. The novelty of our framework is in its ability to *approximately* measure test progress against a criterion without constructing the graph model. In our framework, each test run is monitored and synchronization events are recorded in a trace. When a test run finishes, the trace is analyzed to speculate alternate behaviors that are feasible but not yet covered. The speculated behaviors, combined with other information, are then used to estimate test progress, decide when to stop, and guide the testing process to increase the chance of finding faults in a systematic manner.

II. RACE ANALYSIS

Many factors can contribute to the non-deterministic behavior of concurrent programs. Examples are unpredictable process scheduling, variations in message latencies, and/or the explicit use of non-deterministic constructs. These factors are referred to as race conditions or just races. Race analysis refers to the activity of identifying races and has found many applications in analyzing, debugging, and testing of concurrent programs.

Our framework leverages off the use of race analysis to estimate test progress and guide the testing process. The key observation is that race analysis can be used to speculate alternate behaviors. Figure 1 depicts a trace recorded from an execution of a message-passing program. Due to variations in message latencies, it is possible that the message sent by $s2$ arrives at T2 before that sent by $s1$. Thus, we can speculate that the message sent by $s2$ can be received by $r1$ in another execution. It is said that $s2$ is in the race set of $r1$. On the contrary, we point out that $s1$ is not in the race set of $r2$. This is because if $r1$ receives a message other than that sent by $s1$, then whether or not $r2$ exists depends on how the program is implemented. In [4], Tai described a race analysis algorithm that guarantees no false positive, meaning that every reported race can actually occur in at least one program execution, regardless of how a program is implemented.

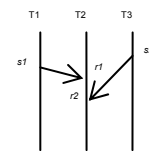


Figure 1: An example execution

III. THE FRAMEWORK

scenario. Bob finishes writing a small multithreaded program and starts to test it non-deterministically. In the beginning, bugs are everywhere, and almost every test run fails. Each time a bug is fixed, the program surely needs to be re-tested. This decision process works well, until he finds that there has been five successful test runs. Then, he ponders, “shall I stop or continue?”

The key issue in the above scenario is that Bob does not have a way to measure test progress. In the sequel, we explain a tentative test strategy, using a message-passing program and the *all-synchronization* criterion. It should be noted that the strategy can also be applied to other types of concurrent programs as well as other testing criteria.

Let P be a message-passing program. A synchronization occurs between a send event s and a receive event r during an execution of P if the message sent by s is received by r . The *all-synchronization* criterion requires all possible synchronizations be covered. Let $ALL_SYN(P)$ be the set of all synchronizations of P . Let t be a trace recorded from an execution of P . Let $SYN_COVERED(t)$ be the set of synchronizations exercised in t . Let $SYN_PREDICTED(t)$ be the set of synchronizations that can be speculated from race analysis of t .

We wish to stress that in order to determine the ALL_SYN set, program semantics needs to be considered. The novelty of our strategy is in its ability to *approximately* measure test progress against the *all-synchronization* criterion without actually deriving the ALL_SYN set.

Let us consider the following strategy, called NT-ALL-SYN, for non-deterministic testing:

1. Initialize T , $SYN_COVERED$ and SYN_MORE to be empty sets.
2. Execute P with a given input non-deterministically. Let t be the trace recorded from the execution. Compute $SYN_COVERED(t)$ and $SYN_PREDICTED(t)$.
3. Perform the following updates: $T = T + \{t\}$; $SYN_COVERED = SYN_COVERED + SYN_COVERED(t)$; $SYN_MORE = SYN_MORE + SYN_PREDICTED(t) - SYN_COVERED$.
4. Repeat 2, 3 and 4 until SYN_MORE gets empty.

Figure 2 shows a simple program consisting of five threads, where $ALL_SYN = \{(s1, r1), (s1, r2), (s2, r1), (s2, r2), (s3, r3), (s3, r4), (s4, r3), (s4, r4)\}$. Note that it is often impractical to compute the ALL_SYN set, which is provided here only as a reference point to explain our test strategy.

<p>T1: $s1$: send(T2, a); T2: $r1$: $x = \text{receive}()$; $r2$: $y = \text{receive}()$; T3: $s2$: send(T2, b); $s3$: send(T4, c); T4: $r3$: $u = \text{receive}()$; $r4$: $v = \text{receive}()$; T5: $s4$: send(T4, d);</p>

Figure 2: An example program

Consider the following scenario. We execute the program non-deterministically, and Q1 in Figure 3 is recorded. $SYN_COVERED(Q1) = \{(s1, r1), (s2, r2), (s3, r3), (s4, r4)\}$, and $SYN_PREDICTED(Q1) = \{(s2, r1), (s4, r3)\}$. (Note that $s2$ is in the race set of $r1$ and $s4$ in the race set of $r3$ in Q1.) We perform the updates in Step 3 so that $T = \{Q1\}$, $SYN_COVERED = SYN_COVERED(Q1)$, and $SYN_MORE = SYN_PREDICTED(Q1)$. Then, we execute the program again, and Q2 in Figure 3 is the trace recorded. $SYN_COVERED(Q2) = \{(s1, r2), (s2, r1), (s3, r3), (s4, r4)\}$, and $SYN_PREDICTED(Q2) = \{(s1, r1), (s4, r3)\}$. Again we perform the necessary updates so that $T = \{Q1, Q2\}$, $SYN_COVERED = \{(s1, r1), (s1, r2), (s2, r2), (s2, r1), (s3, r3), (s4, r4)\}$, and $SYN_MORE = \{(s4, r3)\}$. We execute the program one more time, and Q3 in Figure 3 is the trace recorded. After we perform the necessary computations, $T = \{Q1, Q2, Q3\}$, $SYN_COVERED = \{(s1, r1), (s1, r2), (s2, r2), (s2, r1), (s3, r3), (s3, r4), (s4, r3), (s4, r4)\}$, and $SYN_MORE = \{\}$. The testing process then stops, as SYN_MORE becomes

empty. Now that $SYN_COVERED = SYN_ALL$, we have effectively accomplished the *all-synchronization* criterion.

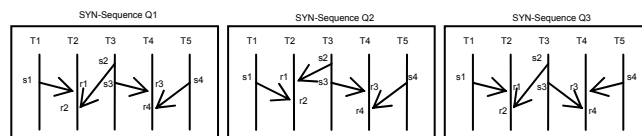


Figure 3: SYN-sequences

The above scenario is possible, but we have to admit that it is also a lucky scenario. To effectively implement the NT_ALL_SYN strategy, several research challenges need to be addressed. First, ideally, we would like to use $SYN_COVERED / SYN_ALL$ to measure test progress. However, the SYN_ALL set is often impractical to compute. Can we use $SYN_COVERED / (SYN_COVERED + SYN_MORE)$ to measure test progress?

Secondly, the stopping condition “*SYN_MORE gets empty*” turns out to work well in the example scenario. However, we cannot expect this to always be true, especially when the control flow is complex. We need to develop advanced stopping conditions so that if they are met, $SYN_COVERED$ is likely to be equal to SYN_ALL .

Finally, what if the size of T does not change while the stopping criterion is still not satisfied? This occurs when some SYN -sequences are repeatedly executed while other SYN -sequences are never executed. We will target on the SYN_MORE set, which contains synchronizations that are feasible but yet to be covered, and develop heuristics to insert noise makers or adjust them if they already exist to increase the chance of making further progress in subsequent test runs.

IV. CONCLUSION

We have described a coverage-based, adaptive framework for non-deterministic testing. The framework exploits the ability of race analysis to speculate alternate behaviors for estimating test progress, deciding when to stop, and guiding the testing process to increase the chance of finding faults by repeated executions.

REFERENCES

- [1] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithread Java program test generation. *IBM Systems Journal*, Vol. 41(1), pp. 111-125, 2002..
- [2] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of the Second Workshop on Runtime Verification (RV)*, Vol. 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.
- [3] C. Yang and L. L. Pollock. Identifying redundant test cases for testing parallel language constructs. *ARL-ATIRP First Annual Technical Conference*, 1997.
- [4] K. C. Tai. Race analysis of traces of asynchronous message-passing programs. *Proc. of the 17th International Conference on Distributed Computing Systems*, pp. 261-268, 1997.