State-Space Analysis as an Aid to Testing

Michal Young

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398

Abstract

Non-determinism makes testing concurrent software difficult. We consider how pre-run-time state-space analysis can be used to aid in testing implementations of concurrent software. State-space analysis techniques have the advantage in principle of exploring all possible execution histories, but they do not verify all properties of interest and in practice they may not accurately model program execution. Combining state-space analysis with testing can partially overcome the weaknesses of each. Using the state-space model in a test oracle is the simpler part: techniques based on classical automata theory are suitable for this. Covering all important non-deterministic executions is harder. We propose a pragmatic method for detecting unexecuted paths that are certainly executable and possibly important.

State-space analysis

A good deal of progress has been made in recent years in improving state-space analysis performance for typical or well-structured programs. We have applied our own Pal tool, which performs hierarchical state-space analysis of Ada-like programs [6,7,8] to some realistic problems (Sanden's furnace design [7], the elevator simulation described in [3]) and some real applications including a portion of the run-time configuration of the Chiron user interface development system. Our experience suggests that current state-space analysis techniques scale up at least far enough to be useful, and certainly far beyond the comprehension of unaided designers and programmers, but what we succeed in analyzing is typically extracted and reworked from the original application code. Testing is required to increase our confidence that the analysis results reflect actual program behavior.

Inducing paths for structural coverage

Structural test coverage criteria for concurrent programs have been proposed by Weiss and by Taylor, Levine, and Kelly [4,5]. These proposals describe criteria for thoroughly exploring non-deterministic execution paths. Their weakness is that the number of such paths grows very quickly with the complexity of synchronization structure in the program to be tested, and it is very costly to determine which of these paths is executable.

We propose a formally weak but pragmatic tactic: An observed execution shows one possible behavior which is known to be feasible. From an observed execution we can infer prefixes of other executions which differ only with respect to non-deterministic choices (e.g., scheduler decisions) and are therefore also known to be feasible. Although this approach does not provide the strong assurances of fault coverage that protocol conformance testing does, those assurances rest on assumptions that do not hold for most concurrent programs [1].

If another (sequential) testing criterion is already in use, it serves as a convenient source of executable paths and corresponding test data. This other method (which could be an arbitrary combination of random, specification-based, and implementation-based testing) "generates" only paths that have already been tested, but this is quite acceptable.We need only assume that this "base method" would be acceptably thorough in the absence of non-determinism. As program testing to some base criterion proceeds, observed execution sequences are used as input to a path induction mechanism. Using a state-space model of the program, this mechanism produces prefixes of an observed path up to a non-deterministic program decision. Each path prefix is known to be executable, and input data to exercise it is already available, so we can safely add an obligation to test different suffixes.

The number of non-deterministic variations on a single executable path is typically very large, and many of them are trivial variations in process interleaving. Since recognizing general races is NP-hard [2], it is not practical to produce additional test obligations for exactly the non-deterministic decisions that can affect program input/output behavior. An interesting alternative is an opportunistic variant of one of the structural criteria from [4] or specification-based test selection from protocol conformance testing, e.g. [1]. One could check observed executions not only for coverage of paths prescribed by one of these techniques, but also for *potential* coverage (variations on an observed execution that improve coverage for one of these other measures).

This research was supported in part by IBM through the IBM/Purdue shared research program, with additional support from the National Science Foundation grant number 91-57629-CCR, and the Advanced Research Projects Agency under grant number MDA-972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of IBM or of the Government, and no official endorsement should be inferred.

References

1. S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Engineering 17*(6), June 1991.
2. R.H.B. Netzer and B.P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems 1*(1), Mar. 1992.
3. D.J. Richardson, S.L. Aha, and T.O. O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th Intl. Conference on Software Engineering*, Melbourne, Australia, May 1992.
4. R. Taylor, D. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Trans. Software Engineermg, 18*(3):206--215, March 1992.
5. S.N. Weiss. A formal framework for the study of concurrent program testing. *Proc. 2nd Workshop on Software Testing, Verification, and Analysis*, pp 106-113, Banff, Canada, July 1988.
6. W.J. Yeh and M. Young. Compositional reachability analysis using of Ada programs using process algebra. Submitted for journal publication; an earlier version appeared in *Proc. 4th ACM Symposium on Testing, Analysis, and Verification*, Victoria, BC, Oct. 1991.
7. W.J. Yeh and M. Young. Redesigning tasking structures of Ada programs for analysis: A case study. SERC-TR-148-P, Dec 1993. Submitted for publication.
8. W.J. Yeh and M. Young. Hierarchical tracing of concurrent programs. *Proc. 3rd Irvine Software Symposium*, pp 73-84, Costa Mesa, CA, April 1993.