An Environment for Testing Concurrent Programs Based on Rewrite-theory Specifications

1710458 Do Minh Canh

Today, software systems are used in various applications where failure is unacceptable. Among are airplanes, vehicles, utilities, telephones, banking & financial systems, commerce, logistics, appliances, houses, and securities. Very important software systems, such as operating systems and the Internet, that have been used as infrastructures are typically in the form of concurrent programs. Major concepts of programming languages that can be used to write concurrent programs emerged in the 1980s and since nearly then studies on testing concurrent programs have been conducted. Arora, et al. have comprehensively surveyed testing concurrent programs. They categorize it into eight classes: (a) reachability testing, (b) structural testing, (c) model-based testing, (d) mutation-based testing, (e) slicing-based testing, (f) formal method-based testing, (g) random testing, and (h) search-based testing. Model checking concurrent programs has been intensively studied, which may be classified into (c) and/or (f). Java Pathfinder (JPF) is such a model checker. Model checking is superior to the other testing techniques in that the former exhaustively checks all possible execution paths (or computations). However, model checking concurrent programs often encounters the notorious state explosion, which has not yet been conquered reasonably well. Therefore, testing techniques for concurrent programs must be worth studying so that they can be matured enough.

For a formal specification S and a concurrent program P, to test P based on S, we can basically take each of the following two approaches: (1) P is tested with test cases generated from S and (2) it is checked that state sequences generated from P can be accepted by S. The two approaches would be complementary to each other. Approach (1) checks if P implements the functionalities specified in S, while approach (2) checks if P never implements what is not specified in S. In terms of simulation, approach (1) checks if P can simulate S, while approach (2) checks if S can simulate P. Approaches (1) and (2) are often used in the program testing community and the refinement-based formal methods community, respectively, while both (1) and (2), namely bi-simulation, are often used in process calculi. This thesis proposes a new testing technique for concurrent programs based on approach (2) mainly because P is a concurrent program and then could produce many different executions due to the inherent nondeterminacy of P.

The proposed technique is basically a specification-based testing one. We suppose that S is specified in Maude and P is implemented in Java. Java

Pathfinder (JPF) and Maude are then used to generate state sequences from P and to check if such state sequences are accepted by S, respectively. Even without checking any property violations with JPF, JPF often encounters the notorious state space explosion while only generating state sequences because there could be a huge number of different states reachable from the initial states, there could be a huge number of different state sequences generated due to the inherent nondeterminacy of concurrent programs and a whole big heap mainly constitutes one state in a program under test by JPF. Thus, we propose a technique to parallelize state sequences generation from P and check if such state sequences are accepted by S in a stratified way. The state space reachable from each initial state is divided into multiple layers. Let us suppose that each layer l has depth d_l. Let d_0 be 0. For each layer l, state sequences $s_0^l, \ldots, s_{d_l}^l$ whose depth is d_l are generated from each state at depth $d_0 + \ldots + d_{l-1}$ from P. Each s_i^l is converted into the state representation $f(s_i^l)$ used in S, where f is a simulation relation candidate from P to S. We conjecture that if S is refined enough, f would be an identity function. There may be adjacent states $f(s_i^l)$ and $f(s_{i+1}^l)$ such that $f(s_i^l)$ is the same as $f(s_{i+1}^l)$. If so, one of them is deleted. We then have state sequences $f(s_0^l), \ldots, f(s_N^l)$, where the number $N + 1$ of the states in the sequence is usually much smaller than $d_l + 1$ because execution units in P are much finer than those in S. We check if each $f(s_0^l), \ldots, f(s_N^l)$ is accepted by S with Maude. The proposed technique is called a divide & conquer approach to testing concurrent programs, which could be naturally parallelized. We have implemented a tool supporting the proposed technique in Java. Some experiments demonstrate that the proposed technique mitigates the state space explosion instances from which otherwise only one JPF instance cannot suffer.

Keywords: testing concurrent programs; specification-based testing; Java Pathfinder(JPF); divide & conquer; Maude; meta-programming; simulation relations.