

Test Suite Consistency Verification

Sergiy Boroday, Alexandre Petrenko, Andreas Ulrich*

Centre de recherche informatique de Montreal (CRIM), *Siemens AG

Sergiy.Boroday@crim.ca, Alexandre.Petrenko@crim.ca, Andreas.Ulrich@siemens.com

Abstract

Test cases are themselves prone to errors, thus techniques and tools to validate tests are needed. In this paper, we suggest a method to check mutual consistency of tests in a test suite

1. Introduction

Despite recent progress in formal verification and other quality improvement methods, testing remains the major quality assurance activity. Thus the quality of software depends on the quality of tests. We address a common problem in the test area: “test the tester”. Unlike most of the previous work, we emphasize mutual consistency of tests, rather than well-formedness of individual tests [1] or adequacy of tests with respect to a formal specification [2, 3, 4] which is not always available in practice.

We model tests using Input/Output Transition Systems (IOTS) [2], also known as Input/Output Automata. Transition systems are mainly used in modeling software, mixed software/hardware systems, and, sometimes, asynchronous circuits. Transition systems often serve as a formal foundation of visual modeling languages such as Message Sequence Charts (MSCs) and UML.

We assume that neither the SUT (System Under Test) nor the tester executing a test can refuse (block) communications. To reduce the number of transitions of a test represented as an IOTS we do not model the fail state reached by the test when unexpected SUT outputs arrive and its incoming transitions explicitly, but assume that SUT outputs that are unspecified in the IOTS test trigger a fail verdict and testing stops. The testing also stops if a deadlock state with a pass verdict is reached. We assume a “slow tester,” meaning that prior sending an input or producing a pass verdict, the tester waits a certain amount of time to observe all possible outputs of SUT or their absence. The absence of any SUT output in a given state is modeled by a designated quiescence output [2]. We assume that the

SUT is a reactive system and not a “generating” one that constantly produces outputs even in the absence of input stimuli. The SUT can be output non-deterministic [2] in the sense that it can produce different outputs in response to an input. Non-determinism occurs when the SUT behavior depends on some inputs or parameters which are beyond the tester’s control. Even if the SUT is output deterministic, the test designer unaware of some implementation choices may resort to abstraction which might result in non-determinism of SUT outputs in the tests.

The paper is organized as follows. Section 2 introduces an IOTS and related notions. In Section 3, test well-formedness and consistency conditions are formulated. A method for verifying test consistency is suggested in Section 4. Section 5 discusses issues related to tests with the inconclusive verdict, concurrent tests, and a refinement of the test consistency relation. Section 6 compares our results with the preceding work and concludes the paper.

2. IOTS

An IOTS L is a quintuple $\langle S, I, O, \lambda, s_0 \rangle$, where S is a set of states; I and O are disjoint sets of input and output actions, respectively; $\lambda \subseteq S \times (I \cup O) \times S$ is the transition relation; and s_0 is the initial state [2]. IOTS L is *deterministic* if λ is a function.

We use the designated output symbol $\delta \in O$ to denote observable quiescence (absence of outputs). In practice it is implemented by a timer of sufficient size for the SUT to stabilize.

A sequence $u \in (I \cup O)^*$ is called a *trace* of IOTS L from state $s_1 \in S$ if there exists a path $(s_1, a_1, s_2)(s_2, a_2, s_3) \dots (s_n, a_n, s_{n+1})$, such that $u = (a_1 \dots a_n)$. For state s of a deterministic IOTS L and trace u , *s-after-u* denotes the state reached by L from state s when trace u is executed.

We use $En(s)$ to denote the set of actions enabled in state s , i.e., $En(s) = \{a \in (I \cup O) \mid \exists t \in S, (s, a, t) \in \lambda\}$. We partition the set of actions enabled in state s into input and output actions, $En(s) = In(s) \cup Out(s)$, where

$In(s) = I \cap En(s)$ is the set of enabled inputs in state s , while $Out(s) = O \cap En(s)$ is the set of enabled outputs. State $s \in S$ is a *deadlock* if no action is enabled in it, i.e., $En(s) = \emptyset$.

Given enabled actions $a, b \in En(s)$, if $b \notin En(s\text{-after-}a)$ then we say that a *disables* b in state s .

Given $L_1 = \langle S_1, I_1, O_1, \lambda_1, s_{01} \rangle$ and $L_2 = \langle S_2, I_2, O_2, \lambda_2, s_{02} \rangle$, the *product* of L_1 and L_2 is the minimal (in terms of the number of states and transitions) IOTS $L = \langle S, I_1 \cap I_2, O_1 \cap O_2, \lambda, (s_{01}, s_{02}) \rangle$ such that $S \subseteq S_1 \times S_2$, and for each state $(s_1, s_2) \in S$, each transition (s_1, a, s_1') of L_1 , and each transition (s_2, a, s_2') of L_2 , there exists a transition $((s_1, s_2), a, (s_1', s_2'))$ of L . The set of traces of a product of several IOTSs is the intersection of the sets of traces of these IOTSs.

3. Test consistency

Defining tests, we take the view of the SUT, meaning that inputs in tests are inputs to the SUT, while outputs of tests are outputs from the SUT. Here a *test* is a deterministic IOTS T , such that a designated quiescence output $\delta \in O$ labels any transition from a state where no input is enabled to a state where no output at all including δ is enabled.

A test is called *well-formed* if

- The only enabled action of the initial state is δ ;
- It is *controllable*, i.e., there is no choice between input and output. Thus, either $Out(s) \neq \emptyset$ or $In(s) \neq \emptyset$, and at most one input is enabled in each state, i.e., $|In(s)| \leq 1$;
- The set of traces is finite.

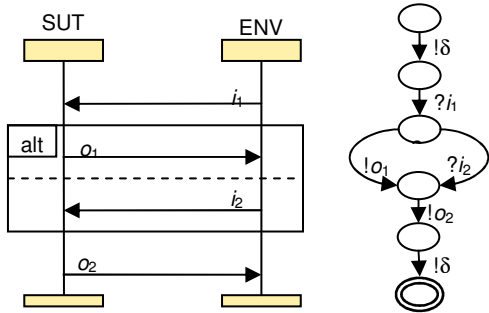


Figure 1. An ill-formed test

An example of an ill-formed (not well-formed) test is shown in the right part of Figure 1. The test is illustrated by an MSC in the left part. The test is ill-formed since it is uncontrollable. Several examples of the well-formed tests are shown in Figure 2.

The trace p is an *unexpected trace* of a test if $p = p_1 o$ is not a trace [2, 3, 5] of the test where p_1 is a trace

of the test, and o is an output, possibly δ . Unexpected traces are traces of the SUT which lead to the fail state of the test. As an example, δi_1 is a trace of test B in Figure 2, while trace $\delta i_1 o_1$ is not and thus is an unexpected trace of B .

Two well-formed tests are said to be *inconsistent* if a trace of one test is an unexpected trace of the other. In other words, two tests are inconsistent if they classify the same SUT behavior as correct and incorrect. Inconsistency indicates that at least one of the tests is either unsound [2, 3], i.e. it produces a fail verdict for a correct SUT, or lax [3], i.e. it tolerates observed erroneous behavior of the SUT.

As an example, $\delta i_1 o_1$ is a trace of test A (Figure 2). At the same time, $\delta i_1 o_1$ is an unexpected trace of test B . Thus, the two tests are inconsistent. One possible reason for this is that one of them is unsound. If we assume however that both tests are sound then test A is lax, namely removal of the transition labeled with o_1 will only improve the fault detection power of the test. Tests B and C are also inconsistent, moreover they never produce the same verdict for any given SUT. That means at least one of them is necessarily unsound.

Well-formedness of a test is rather straightforward to check; hereafter we discuss only well-formed tests. A method for checking test consistency is suggested in the following section.

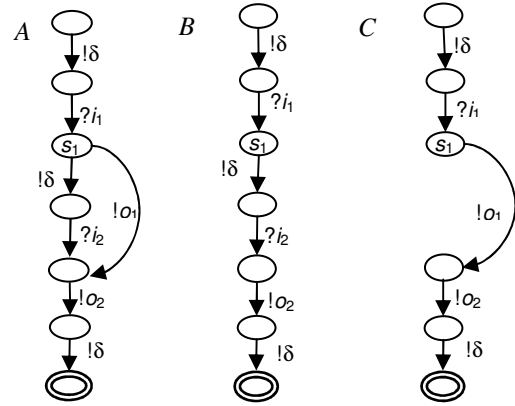


Figure 2. Well-formed tests

4. Test suite consistency verification

4.1 Verification of test consistency

Proposition 1. T_1 and T_2 are inconsistent if and only if at least for one state (s_1, s_2) of their product different outputs are enabled in the state s_1 of T_1 and state s_2 of T_2 , i.e., $Out(s_1) \neq Out(s_2)$.

Thus, consistency of tests can be verified by computing their product. For example, the product of inconsistent tests A and B shown in Figure 2, contains state (s_1^A, s_1^B) such that $Out(s_1^A) \neq Out(s_1^B)$.

Obviously, consistency analysis makes sense only for tests executed from the same state. While in theory it is common to assume that each test of a test suite has to be executed from the SUT's initial state, it is often not the case in practice. The next section discusses consistency verification for tests with dependencies.

4.2 Consistency of test suite

The use of test suites with dependencies promotes modularity, reusability and allows one to better structure the suites. Otherwise, a test suite could be a collection of very long tests, which share significant fragments. To avoid redundancy, tests are specified to be executed in different SUT states. In practice, these starting states are often described in the form of preconditions. A precondition is fulfilled if execution of another test terminates in the start state of the new test. In this case we say that the two tests depend on each other. The first test serves as a preamble of the second test. In some cases, several alternative preambles could be associated to a test which might result even in cyclic dependencies, e.g., a “reset” test could be used to move a correct SUT back into its initial state. The strategy to execute such tests is not necessarily known at design time of the test suite. Thus test suite consistency must be verified irrespective of the test execution strategy.

Here we present a formal model to describe such preconditions in terms of an IOTS by considering a deadlock state of one test as a precondition to one or several other tests. Note that in the case when a test has several preconditions they are considered as alternatives.

A *test suite* is a tuple $\langle \mathbf{T}, I, O, S^{\text{fin}}, Pre, s_{\text{init}} \rangle$, where $\mathbf{T} = \{ \langle S_1, I, O, \lambda_1, s_{01} \rangle, \dots, \langle S_n, I, O, \lambda_n, s_{0n} \rangle \}$ is the set of tests over the sets of inputs I and outputs O , with disjoint state sets, S^{fin} is the set of deadlocks in the tests, $Pre: \mathbf{T} \rightarrow \{s_{\text{init}}\} \cup S^{\text{fin}}$ is the precondition function which describes test dependencies.

Informally speaking, a test suite is *inconsistent* if at least one pair of composite tests built by appending tests according to the precondition function is inconsistent.

In order to verify the consistency of a test suite, we introduce the notion of a Test Suite IOTS (TSI) which describes input/output sequences defined by the test suite. The set of states of a TSI is the union of all non-initial states of the tests contained in the test suite and

s_{init} which is the initial state of TSI. The transition relation of the TSI is the minimal relation, such that

- For each transition (s, a, s') of each test T_i which does not involve initial states of tests, there is a corresponding transition (s, a, s') in the TSI, and
- For each transition (s_{0i}, a, s) from the initial state of each test T_i , and each $s' \in Pre_i$, there exists a transition (s', a, s) in the TSI.

Informally, the Test Suite IOTS is obtained by connecting “disconnected” test IOTSs namely by attaching each test to states that constitute its precondition. Since a test precondition could contain several deadlock states, the initial state of each test is duplicated accordingly and merged with the deadlock states of the precondition. Now to check test suite consistency, the self-product of the TSI can be constructed. Inconsistency manifests itself in the presence of at least one state (s_1, s_2) of the product, where the set of outputs enabled in s_1 differs from the set of outputs enabled in s_2 (note that quiescence is treated as an output).

4.3 Weak and strong inconsistency

In Section 3, we have shown that test inconsistency does not necessarily imply unsoundness. This observation motivates us to refine the inconsistency relation.

Two inconsistent tests T_1 and T_2 are called *weakly* inconsistent if they share at least one trace $p\delta$ such that $In(T_1\text{-after-}p\delta) \cap In(T_2\text{-after-}p\delta) = \emptyset$, i.e. the two tests disagree on some output and agree on an alternative output such that either a complete trace (leading into deadlock) of one test is a trace of another test or two traces of these two tests have a common proper prefix extended with different inputs. This means that there exists an SUT which passes both weakly inconsistent tests since after a common trace each test allows several output reactions and the two tests agree on some of them. Inconsistent tests which are not weakly inconsistent are called *strongly* inconsistent. No SUT passes two strongly inconsistent tests, thus at least one of them is necessarily unsound. The following statement gives a criterion to check whether two inconsistent tests are weakly or strongly inconsistent.

Proposition 2. The product of weakly inconsistent tests contains a deadlock state with an incoming δ transition, while the product of strongly inconsistent tests does not contain such a state.

5. Discussions

5.1 Tests with inconclusive verdicts

Sometimes, along with pass and fail, the inconclusive verdict is allowed for non-deterministic systems. Deussen and Tobies [1] consider a test purpose (an abstract, possibly non-deterministic specification of a test) to be ill-formed if it can yield different (pass, fail, or inconclusive) verdicts for the same trace. However, we do not distinguish between the pass and inconclusive verdicts here as these verdicts apply only to a correct SUT behavior [6]. That is our test inconsistency detection approach still applies to tests where some deadlock states are considered inconclusive simply by discarding the difference between usual (pass) and inconclusive test verdicts. We believe that in a correct test suite two different tests can produce inconclusive and pass on the same trace as long as the inconclusive verdict does not indicate incorrect SUT behavior.

Well-formedness of tests with the inconclusive verdict however needs further refinement. An additional property in this context could be the reachability of at least one pass state.

5.2 Weak and strong inconsistency

The definitions of weak and strong inconsistency could be generalized for arbitrary sets of tests. However, their detection requires construction of a (multi) product of all given tests which can be computationally difficult due to state explosion.

5.3 Concurrent tests

The suggested test consistency definitions do not readily apply to concurrent tests. For example, test stimuli can be fed or concurrently by independent test components either to speed up testing or to test the SUT behavior under a concurrent load.

Tests in Figure 3 occur as inconsistent according to the definition in Section 4, as quiescence transitions occur only at the beginning and end of the concurrent test, while the sequential test has one more quiescence transition. However, the latter test is just a sequentialized version of the concurrent one.

To allow both concurrent and sequential versions of a test to appear in the same test suite, the consistency definition has to be relaxed. One possible way of achieving this could be to drop explicit modeling of the absence of outputs by replacing quiescence transitions by traditional internal transitions [2], and to define

inconsistency of tests based on the existence of a trace p_1ip_2 , where p_1i is a common trace of both tests, i is an input, p_2 is an output sequence such that it takes only one test to reach a state where no output is enabled. If there exists a sequence p_1ip_2 , where p_1i is a common trace of both tests, i is an input, p_2 is an output sequence, such that either p_1ip_2 is not a common trace of the two tests or it can be extended with outputs only in one of the tests.

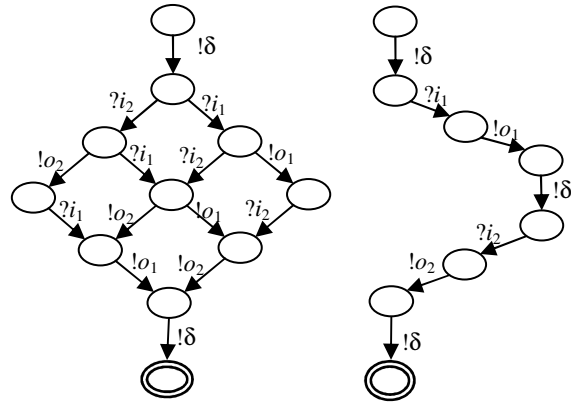


Figure 3. Concurrent and sequential tests

6. Related work and conclusions

We suggested a method for verifying a test suite with inter-test dependencies (preconditions) which, unlike previous works [3, 4, 6], does not rely on an SUT specification. At the same time we showed how inconsistencies, detectable without any use of an SUT specification relate to test flaws such as unsoundness and laxness which were previously defined and detectable only based on a given SUT specification [3].

A close approach is developed by Deussen and Tobies [1] for verifying tests, where a given test purpose should not contain ambiguities related to the test verdicts. However, we do not consider test purposes, while tests are deterministic and thus unambiguous. Instead, we address ambiguities among different tests of a test suite. We also demonstrate that certain cases of inconsistencies are not necessarily due to flaws in the tests.

Unlike the model checking approach to decide on test consistency [7] we address generic rather than application or domain specific test consistency requirements.

7. References

- [1] P. Deussen, S. Tobies, “Formal Test Purposes and the Validity of Test Cases”, *FORTE 2002*, pp. 114-129.
- [2] J. Tretmans, “Test Generation with Inputs, Outputs and Repetitive Quiescence”, *Software Concepts and Tools*, 1996, pp. 103-120.
- [3] C. Jard, T. Jeron, and P. Morel, “Verification of Test Suites”, *TestCom 2000*, pp. 3-18.
- [4] K. Naik, B. Sarikaya, “Verification of Protocol Conformance Test Cases Using Reachability Analysis”, *J. of Systems and Software*, 19(1), 1992, pp. 41-57.
- [5] J. Huo, A. Petrenko, “Covering Transitions of Concurrent Systems through Queues”, *ISSRE 2005*, pp. 335-345
- [6] G. Bochmann, D. Desbiens, M. Dubuc, D. Ouimet, and F. Saba, “Test Result Analysis and Validation of Test Verdicts”, *Protocol Test Systems*, 1990.
- [7] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, and H. Ide, “Automated Regression Testing of CTI-Systems”, *IEEE European Test Workshop*, 2001, pp. 51-57.