Evaluation of test coverage for embedded system testing

Jinsong Zhu, Son T. Vuong
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4

Samuel T. Chanson
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

In this paper we propose an original approach to the evaluation of test suites for embedded system testing, where the implementation under test (IUT) is embedded in a composite system as a component module. We define a coverage measure based on the identification of the IUT within the test context with respect to observational equivalence at the composite system level. The problem of limited IUT controllability and observability caused by the test context is handled when computing the coverage. The approach is purely functional and only assumes a general fault model where the number of states in the IUT is upper bounded. A tool has been developed and an example is given to illustrate and validate the approach.

Keywords

Protocol testing, embedded test, test in context, fault coverage

1 INTRODUCTION

With the development of distributed component computing, it has become increasingly important to test components that are embedded in a composite

A. Petrenko et al. (eds.), *Testing of Communicating Systems*

system. Often this has to be done at the composite system level due to the difficulty in isolating an embedded component that is fully integrated into the system. All components other than the one to be tested constitute the *test context* [7]. In such an environment, testing is performed by applying test suites at the system level in order to exercise the behavior of the target component since the interfaces of the embedded module are not directly accessible by the tester. The correctness of the component can only be inferred from the global system behavior, *i.e.,* system input and output events. Internal events between components act as constraints on the global behavior and are not directly controllable or observable. This type of testing is known as *embedded testing* as in the International Standards Organization (ISO) conformance testing framework for communication protocols [7].

Most work in the area of protocol testing has mainly been on single component testing where each component of the system is tested in isolation. Effectiveness of such tests can be evaluated in various ways, ranging from fault simulation [4, 15], structural analysis [17], to faulty machine identification [16, 18].

Recently some work aiming at testing embedded components (also called testing in context) has appeared [13, 14, 12, 9]. The work so far has mainly been on defining fault models and generating effective test suites with respect to given fault models. However, the problem of evaluating the coverage of a given test suite for embedded component testing has not been addressed. The objective of this paper is to propose a solution to this problem.

The traditional fault simulation method as used in fault coverage evaluation for isolated machines [4, 15] can be adapted for use in embedded systems by defining an appropriate fault model as in [13]. However, in many complex systems detailed fault classes are unknown or are difficult to be established. It is simply impossible to explicitly construct all faulty machines in a realistic situation as pointed out in [13]. We therefore propose a new approach to the evaluation of test coverage for embedded system testing that is based on faulty machine identification by given external behavior, *i.e.,* a test suite. No detailed fault classes are needed. In the approach, the components of a system are modeled as two communicating finite state machines (FSMs), one being the test context and the other being the IUT. The test context is assumed to be faultless. We identify the IUT component (an FSM) using both reachability computation and constraint based search, the latter has been effectively used in our earlier work on the identification of single FSMs [18]. The test coverage of the test suite is measured by the number of FSMs that are not conforming to the component specification. The approach increases the computing efficiency in two ways: 1) The global identification is avoided; 2) The constraint based search is optimized by domain reduction and backjumping. The approach can be readily extended to enhance the quality of test suites by generating additional test cases

The rest of the paper is organized as follows. Section 2 gives an overview

of the general principles of embedded system testing and test architectures. Section 3 describes our approach of coverage measure in embedded testing. The next two sections provide a more formal description of the algorithm followed by an illustrative example, respectively. We close with a discussion of the proposed approach and some future work.

2 EMBEDDED TESTING ARCHITECTURE

We model a composite system as a set of communicating FSMs. Figure 1 shows a test architecture that depicts such an environment.

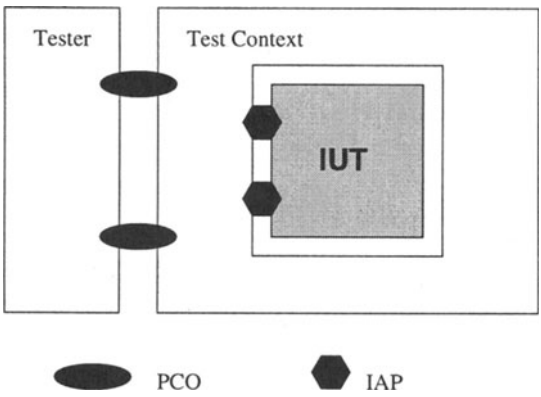


Figure 1 Test architecture for embedded component testing

Major components of the architecture are (see [10] for details):

- a tester;
- an implementation under test (IUT);
- test context;
- points of control and observation (PCO);
- implementation access points (IAP).

The tester contains a test engine that executes test suites and determines whether the IUT (the targeted component, also called Module Under Test, or MUT for brevity) conforms to its specification by observing system responses at the PCOs. The test context is a set of FSMs that are not being tested; instead they act as an environment in which the IUT runs, and which interacts with the IUT through the IAPs. The IAPs are not directly controllable or observable by the tester.

An example of embedded module is the Service Specific Connection Ori-

ented Protocol (SSCOP) used in the B-ISDN ATM Adaption Layer (AAL) [8]. SSCOP is embedded in the AAL, on top of the Common Part Convergence Sublayer (CP-AAL) and below a Service Specific Coordination Function (SSCF) module. Testing of SSCOP needs to be an embedded testing if it is only accessible via the CP-AAL and SSCF.

3 TEST COVERAGE MEASURE

As mentioned before, our coverage measure is based on machine identification. The number of faulty machines that cannot be detected by a test suite reflects the effectiveness of the test suite. We will first give an overview of the measure as applied to single machine identification where the tester has full control of the IUT (*i.e.*, has direct access to its I/O ports). The measure is then extended to embedded component testing.

3.1 The machine identification approach

Machine identification consists in deducing a finite state machine from its input/output sequences which are samples of its external behavior. The issue was initially studied in the area of sequential circuits [11, 2, 5] and later used in communication protocol testing [16, 18, 3]. Figure 2 shows the different sets of machines that can be defined according to this method: S is the specification machine, SCM (the area of the inner-most circle) is the set of machines that conform to S, and TCM is the set of machines that are compatible with the test suite, *i.e.*, they are perceived as correct by the test suite. The shaded area thus represent those machines that do not conform to the specification but cannot be detected by the test suite. It is a measure of the *inefficiency* of the test suite. The coverage, or the *goodness* of the test suite, can be defined as the inverse of the size of the shaded area.

Definition 1 (Coverage measure) *Let T be a given test suite, S the specification, and n the upper bound of the number of states in any implementation of S. Let K_{TCM} be the number of T-conforming machines for S, and K_{SCM} is the number of S-conforming machines. The test coverage of T is defined as:*

$$Cov(S, T, n) = \frac{1}{K_{TCM} - K_{SCM} + 1}.$$

The "1" in the denominator is necessary since it is possible that $K_{TCM} = K_{SCM}$ which should be interpreted as complete coverage instead of infinite coverage, which does not make good mathematical sense. The definition also

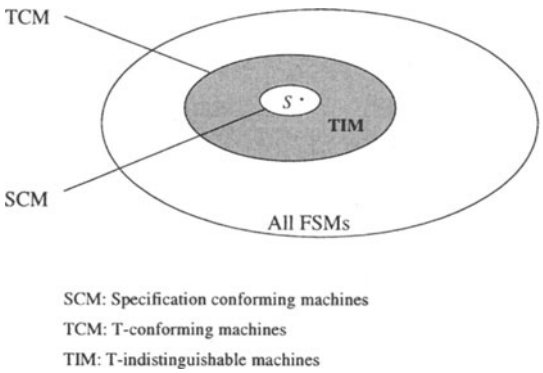


Figure 2 Machine identification with respect to a test suite

makes sure that if a test suite subsumes another one, it must have a higher coverage value.

To compute Cov, we need to:

- calculate K_{TCM}, i.e., to identify the number of machines that accept the test suite, and
- determine how many of these T-conforming machines actually conform to the specification.

To calculate K_{TCM}, identification and construction of the T-conforming machines are necessary. Traditional methods such as [11, 2] may be employed. Assumptions are usually made to make the problem solvable, First of all, the space of all FSMs must be limited since otherwise K_{TCM} may be infinite, causing a zero fault coverage for any test suite. A widely used assumption is the "upper-bound state" assumption where faults in the IUT can only increase the number of states to a predefined boundary. The upper bound should also be relatively small to make it manageable in practice, since the space of FSMs grows exponentially with the number of states. Secondly, in order to construct the T-conforming machines in a deterministic way, the IUT must be modeled as a deterministic finite state machine, using either the Moore or Mealy model (the latter is commonly used in communication protocol modeling).

The identification approach has the major advantages of generality and independence of detailed fault models such as output faults, head and tail state faults, state transfer faults, and their combinations. However, the identification process typically introduces high computational cost as the identification problem is known to be NP-complete with respect to the number of states [6]. In [18], heuristics developed in artificial intelligence have been employed in an effort to reduce the computing time and it has been shown to be effective for some moderate-size protocols [18]. This technique should be even more

feasible at higher levels of abstraction for test design where the number of protocol states would be smaller. We will use this AI based method in our coverage calculation for embedded system testing.

3.2 Identification of embedded machines

In embedded system testing, the coverage could have been calculated by identifying the whole global system. However, this approach would not be interesting because it may run out of steam in handling a complex composite system and also violates our motto of "divide and conquer". Besides, when we assume the test context is faultless (which is perfectly reasonable in embedded system testing), it is simply unnecessary to identify the global system because a lot of energy would be wasted in identifying possible faults in the context.

To calculate the coverage in embedded system testing, we assume that only the embedded component (MUT) is faulty, the test context being thoroughly tested or for our purpose being simply faultless. The MUT is identified by the test suite applied to the whole system (see below). This approach avoids the identification of the global system and allows us to focus on the MUT for a better coverage.

After identification of the MUT, comparison must be made to determine whether it conforms to its specification. As explained in [13, 14], this comparison must be done at the composite system level, *i.e.*, the conformance of MUT must be defined by the global behavior after composition with its test context. This is because with limited controllability and observability of the MUT, we can only determine the conformance by its globally observable behavior. If an MUT does not conform to its specification at the component level, but the global behavior of the MUT in composition with the context conforms to the composition of its specification and the context, we can only conclude that the MUT works correctly *in that context*. This is often called *equivalence in context* [13]. The observational equivalence relation [1] applies in this situation and is used for determining the equivalence in context.

Besides the equivalence issue, we also need to impose the so-called *I/O ordering constraint* [13]. This constraint requires that the next input is submitted to the composite system only after it has produced an output in response to the previous input. This allows the system to have a finite number of steady states. Also, in order for the composition machine (the MUT and the context) to be valid, the product of MUT and context should not contain livelocks.

Since a test suite may consist of many test cases each starting from the initial system state, a special input *reset* is used to bring the system to its initial state. This initial state is a combination of the initial states of the context machine and the MUT. Whenever *reset* is encountered, both the context and MUT are set to its initial state, and no output (or a *null* output) will be produced. The faults in the IUT will not affect this capability.

Let M_I be the embedded MUT module to be identified given its specification S, the test context C, and the test suite TS. We consider TS as composed of a linear sequence of test steps (I/O pairs). If there are many test cases, they will simply be concatenated with *reset/null* to form the test suite.

Our approach in identifying M_I is to first build a test tree for M_I, which represents all sampled behaviors of M_I as excited by the test suite. This test tree is then collapsed for the identification of M_I using the approach proposed in [18]. We follow the following steps to build the test trees:

1. Apply a test step x/y (an I/O pair) in the test suite to the system. Since C is known, reachability computation can be performed until the input reaches M_I (we do not exclude the case where an input is directly applied to an embedded module by bypassing C; this is done simply by skipping the reachability computation part). For example, an input x applied to the system via C may produce an internal event in C as an input to M_I. A special case in this step is the *reset* signal. No reachability computation is needed; we simply reset the context machine and move to the root of the current test tree.
2. When an input reaches M_I, we start to construct a test tree for M_I. At this point, the output caused by that input from C may be any possible event that is allowed in M_I's output alphabet. Try each possible event according to the next step.
3. For each possible event o_p, continue the reachability computation of Step 1, until a global output is produced. If, during the reachability computation, an output of the context is an input back to M_I, Step 2 will be called. This will lead to a recursive execution of Steps 2 and 3. When a global output is finally produced, it is compared with the output of the test step. If they are different, a contradiction is found and hence o_p is ruled out. Repeat this step for the next possibility until a global output compatible with the test step is observed or no possibility is compatible. If a compatible output is found, the test tree goes one step farther with a label of i/o, where i is the internal input to M_I excited by the global input, and o is the compatible output. If no compatible output can be found, backtrack to the previous test step. The current test tree is also "rewound" accordingly.
4. If all test steps have been processed, we have obtained a test tree for M_I. At this point, we backtrack to the previous test step, with C and the test tree being set to their respective previous states, and continue searching for the next output possibility with respect to these states. This step is repeated until all output possibilities have been exhausted, resulting in all test trees for M_I.

If the test suite was correctly generated for S, we will end up with at least one test tree for M_I; otherwise the algorithm will produce no test trees which would indicate an incorrect given test suite. Figure 3 gives a simplified

illustration of this process. Suppose a test step consists of x/y. The global input x causes a transition x/a in C, in which a serves as an internal input to M_I, which outputs y. An edge labeled a/y in M_I's test tree is then obtained. The dotted line from M_I to C indicates a possibility where M_I produces an output to C in response to its input. In general, a finite number of events may be exchanged between C and M_I before a global output is produced. Due to the assumption that there is no livelock between C and M_I, a global input will always cause a global output after its propagation within the system. However, this assumption may not hold for a faulty IUT when we explore all output possibilities of M_I. In such a case, we will terminate the algorithm after a predefined number of interactions between C and M_I, which should be large enough to signify a livelock. More sophisticated mechanism in detecting a livelock on-the-fly requires further investigation.

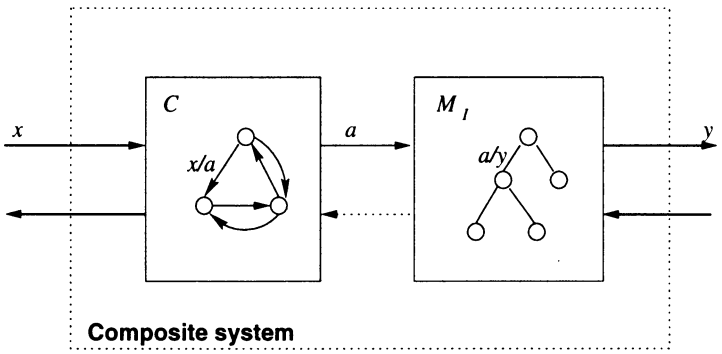


Figure 3 Test tree construction for embedded component

With the test trees constructed, the identification of M_I can be solved as a single machine identification problem using our tool COV as presented in [18]. Each test tree is fed to COV to build the corresponding M_I(s).

In order to determine whether an M_I is a conforming implementation, the final step is to compute the composition of C and M_I, denoted as $C \circ M_I$ [13]. It can be obtained from the product of C and M_I ($C \times M_I$) by hiding the internal actions since observational equivalence is used in our testing. The assumption that $C \times M_I$ has no livelock assures the existence of $C \circ M_I$. The composition is then compared with $C \circ S$ for observational equivalence. Those equivalent in the global behaviors are counted as K_{SCM}. This completes our calculation of K_{TCM} and K_{SCM}.

4 ALGORITHM

This section presents the above algorithm in more formal notations. Our formal model is the familiar Mealy machine frequently used in communication protocol modeling. A deterministic FSM is represented by a quintuple $M = \langle Q, X, Y, \delta, \lambda \rangle$, where Q, X, Y are the internal states, input alphabet and output alphabet respectively. $\delta : Q \times X \rightarrow Q$ is the next state function and $\lambda : Q \times X \rightarrow Y$ is the output function. The functions δ and λ can be extended to handle an input sequence $\sigma = x_1 x_2 ... x_k$ in the usual way: $\delta(q_1, \sigma)$ is the final state after σ is applied to state q_1, and $\lambda(q_1, \sigma)$ denotes the corresponding output sequence. That is, $\lambda(q_1, \sigma) = y_1 y_2 \cdots y_k$ where $y_i = \lambda(q_i, x_i)$ and $q_{i+1} = \delta(q_i, x_i)$ for $i = 1, ..., k$, and $\delta(q_1, \sigma) = q_{k+1}$.

A test case of length k is an element of $(X \times Y)^*$ and is denoted by $\tau_k = \langle x_1/y_1, ..., x_k/y_k \rangle$, which satisfies $\lambda(q_1, x_1 x_2 \cdots x_k) = y_1 y_2 \cdots y_k$. The special test step *reset/null* is given by $r/-$ for short. If a test suite contains a set of test cases each starting from the initial state, it can be considered as a linear test sequence formed by concatenating the test cases with $r/-$. In the following algorithm, we will not make distinction between a test suite and the test sequence obtained by concatenating the test cases in the test suite.

The inputs to our algorithm are the initial state of the MUT to be identified, the initial state of the context, and the test suite. In other words, we build our test tree starting from the initial state of the system. The output is the set of test trees for the MUT, obtained by applying the test suite to the system in the given test context. These trees are then used to infer the MUT as per the single machine identification algorithm [18]. The final composition of the inferred machines and the context is done to single out conforming solutions.

Figure 4 gives the algorithm for constructing the test tree. It is a recursive procedure that processes the test suite one test step each time. The exceptional case where a livelock may exist is not shown.

The first line checks if the whole test suite has been processed. If so the test tree has been constructed and is saved, and the algorithm backtracks to search for other solutions. Line 2 handles the special case of the reset signal, resulting in the initialization of the context as well as the current test tree. The current tree node points to the root. Line 5 applies reachability computation (RC) from the global input until the MUT module M_I is reached, which means an internal input for M_I is produced from the global input. In case where a global output is generated before M_I is reached, *i.e.*, this test step does not really test any part of M_I, the procedure backtracks to the next test step without adding any edge to the test tree. Otherwise, the algorithm proceeds to line 7 to search for an output symbol for a potential edge in the test tree. This is done by analyzing all possible outputs for M_I with regard to the present internal input. In Line 8, for each output possibility, if it is a system output and it is consistent with the output given in the test step, we have found a consistent edge for M_I. If it is a context input instead, the algorithm goes

Algorithm: Test tree construction for M_I.
$\sigma = \langle \sigma_1, ..., \sigma_n \rangle$: the test suite, where $\sigma_i = x_i/y_i$ is the i^{th} test step;
v_i: represents tree node i;
q_i: represents state i of the context;
TT: the test tree being constructed.

```
boolean TestTree(σ_i, v_i, q_i)
{
1. if (σ_i = NULL) { save TT; return(FALSE);}
2. if (σ_i = r/−) { v_i ← v_0; q_i ← q_0; }
3. else {
4.     in ← x_i;
5.     RC from in to M_I; Context reaches q_j with input z;
6.     if ( (λ(q_j, z) = a_i) is input to M_I) {
7.         for b_i ∈ Y do {
8.             if (b_i == y_i) { add edge v_i --a_i/b_i--> v_j to TT; break; }
9.             else if (b_i is input to C) { in ← b_i; goto Line 5 };
10.        }
11.        if (no compatible y found for all b_i) return(FALSE);
12.    }
13.    else if (a_i is system output and a_i ≠ y_i) return (FALSE);
14. }
15. return TestTree(σ_{i+1}, v_j, q_j);
}
```

Figure 4 Test tree construction algorithm

back to Line 5 for further reachability computation. If all failed, we go on to try the next output b_i. If no compatible b_i can be found, which indicates an inconsistent b_i in a previous test step, the algorithm backtracks. Line 15 repeats the processing for the next test step.

5 TOOL AND EXAMPLE

To validate the approach, an experimental tool called ECOV (Figure 5) has been implemented in C on a Linux 2.0 Pentium machine. It is a tool set that consists of tools for test tree construction (TTREE), machine identification by the test tree (COV), and a composition algorithm (COMP) that checks the observational equivalence in context. In the tool set, COV was developed in [18] and is integrated into ECOV via I/O files. ECOV takes as input a test suite, a test context, and the MUT specification, and produces a coverage value and non-conforming machines which may be used for further analysis and test suite enhancement.

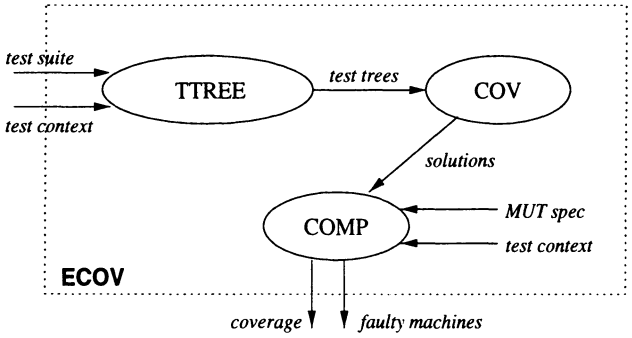
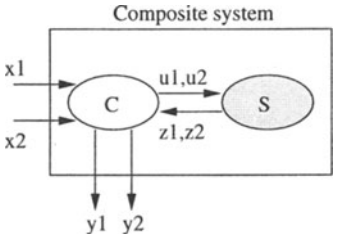
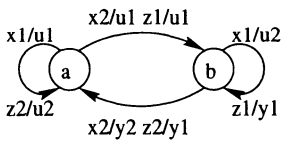


Figure 5 ECOV tool set

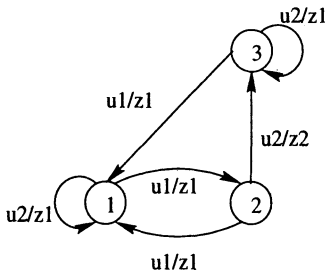
To illustrate the approach, we use the simple example in Figure 6 which is taken from [13]. The test context C takes inputs x_1, x_2 from the system environment and internal input z_1, z_2 from the component under test, and generates system outputs y_1, y_2. C also generates internal outputs u_1, u_2 as inputs to S. The composed machine $S \circ C$ is generated by determinizing the product machine $S \times C$ under the I/O ordering constraint.



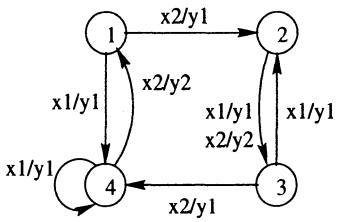
(a) System under test



(b) Test context C



(c) Component S



(d) Composed machine SoC

Figure 6 Example composite system

We use the following test suite from [12] for the component S. This is a complete test suite with respect to the fault model where faults in S's implementation do not increase the number of states:

```
T = {
   r/- x1/y1 x1/y1 x1/y1 x1/y1 x1/y1 x2/y2
   r/- x1/y1 x1/y1 x1/y1 x2/y2
   r/- x1/y1 x1/y1 x2/y2 x2/y1 x1/y1 x2/y1
   r/- x1/y1 x1/y1 x2/y2 x2/y1 x2/y2
   r/- x1/y1 x2/y2 x1/y1 x1/y1 x2/y2
   r/- x1/y1 x2/y2 x1/y1 x2/y2
   r/- x2/y1 x2/y2 x1/y1 x1/y1 x2/y1 x1/y1 x2/y2
   r/- x2/y1 x2/y2 x1/y1 x1/y1 x2/y1 x2/y2
   r/- x2/y1 x1/y1 x1/y1 x1/y1 x2/y1
   r/- x2/y1 x1/y1 x1/y1 x2/y2
   r/- x2/y1 x1/y1 x2/y1 x1/y1 x1/y1 x2/y2
   r/- x2/y1 x1/y1 x2/y1 x2/y2 x2/y1
   r/- x2/y1 x2/y2 x1/y1 x2/y2
}
```

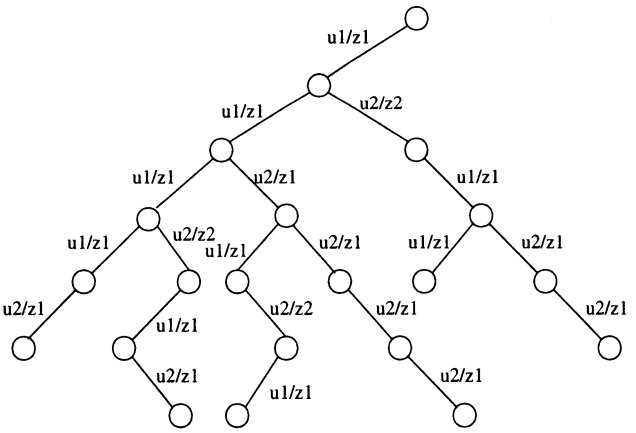


Figure 7 Test tree generated for the embedded component

Our tool yields a test tree for S as shown in Figure 7. This tree is further fed into our tool COV to be collapsed into FSMs. COV produces 19 FSMs, one of which conforms to S at the module level, while the rest, although do not conform to S, are found to be observationally equivalent with S within context C, *i.e.*, the composition of those FSMs conform to the composition of S and C. Therefore, there are no non-conforming solutions at the composite

system level, which leads to the conclusion that T is indeed complete with respect to the embedded testing architecture.

Figure 8 shows two of the generated FSMs that do not conform with S per se but are conforming at the system level. The first one has only two states, yet it is considered conforming when tested in context. This allows us to discover the smallest FSM that implements S without causing system behavior discrepancy. The second one differs from S wildly but still exhibits observational equivalence at the system level.

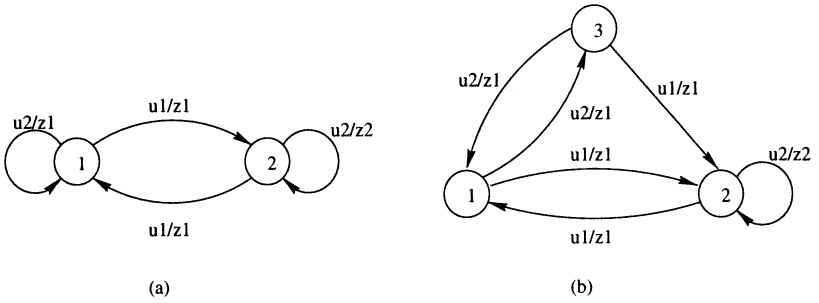


Figure 8 Two FSMs observationally equivalent with S

6 DISCUSSION

So far we have focused on the case where the test context is modeled as a single finite state machine. However, the approach itself does not restrict us only to that situation.

Consider the more general case in which the composite system is composed of multiple modules, and only one of the modules is to be tested as an embedded module (while all other modules are considered faultless). It is obviously unnecessary to compose all the faultless modules to produce a test context of single machine, since some modules may not reach the MUT at all.

Our approach can be easily adapted to this multi-module case while avoiding the costly composition of all text context modules. The structure of the algorithm in Figure 4 does not have to be altered; the only part that is affected is the reachability computation part. When a reachability computation from system input to MUT input is needed, the single module test context is simply replaced by the multi-module one with internal connection between the modules being taken into account. These internal connections are used as constraints on possible interactions between modules, which allows the reachability computation to be performed only for the modules that can reach to the MUT. This should increase the computing efficiency in many cases than if all context modules are simply composed. The backward reachability compu-

tation from MUT output to system output can be performed similarly. The algorithm generates test trees for the MUT just as in the single module case.

Another issue worth discussion is the generation of test suite with complete fault coverage based on the machine identification approach. Our approach supports this by adding extra test cases that distinguish the T-indistinguishable machines (TIMs) (Figure 2). In the normal test environment where the MUT is directly accessible (non-embedded case), this can be done by generating distinguishing test sequences for the TIMs. Algorithms exist that solve this problem.

In embedded system testing, similar approach can be adopted for generating the distinguishing sequences. However, these test sequences must be "externalized" to the system input and output in order for the test sequence to be executable. Fortunately, Petrenko et al have already developed a method for the externalization process [12]. This should allow us to generate complete test suite after coverage evaluation.

7 CONCLUSIONS

We have presented our work on fault coverage evaluation for embedded system testing. The approach is based on the faulty machine identification method that has been successfully employed in direct single module (as opposed to embedded module) testing. By generating test trees for the embedded module, faulty machines can be constructed and therefore fault coverage can be computed. The determination of conformance is done at the composite system level due to the limited observability and controllability imposed by the test context.

The approach effectively extends the single machine identification method into the area of embedded module testing. It inherits the advantages of the method such as its generality and independence of detailed fault classes and their combinations (only the most general fault model is needed where the number of states in an IUT is upper bounded). As a coverage evaluation tool, it not only verifies a given test suite as a complementary of test generation, but can also be used to augment a test suite (by generating additional test steps that detect the faulty machines). The main drawback is higher computational complexity in collapsing the test trees.

We have developed a set of tools that accomplishes the coverage evaluation. The tools have been applied to a simple example as presented in Section 5. The example serves to demonstrate and validate our approach. However, as future work, real protocols such as SSCOP should be used to test the effectiveness of the approach. Moreover, the tool set should be integrated in a more coherent way, since currently the tools are linked via files. For example, the test trees generated by TTREE (Figure 5) are written to a file which is then read by COV. It would certainly be more efficient to directly collapse a test tree once it is generated.

8 ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their very helpful comments.

9 REFERENCES

[1] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[2] A.W. Biermann and J.A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6), June 1972.

[3] G.v. Bochmann, A. Petrenko, and M. Yao. Fault coverage of tests based on finite state models. In *Protocol Test Systems VII*, Japan, 1994.

[4] A. Dahbura and K. Sabnani. An experience in estimating fault coverage of a protocol test. In *Proc. IEEE INFOCOM*, 1988.

[5] E.M. Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972.

[6] E.M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[7] ISO/IEC 9646. *Information Technology - Open Systems Interconnection - Conformance testing methodology and framework.* March 1991.

[8] ITU-T Recommendation Q.2110. *B-ISDN ATM Adaption Layer - Service Specific Connection Oriented Protocol (SSCOP).* ITU Telecommunication Standards Sector, Geneva, 1994.

[9] Luiz Paula Lima Jr. and Ana R. Cavalli. A pragmatic approach to generating test sequence for embedded systems. In *IFIP 10th Int. Workshop on Testing of Communicating Systems*, Cheju Island, Korea, September 1997.

[10] JTC1/SC21/WG1/Project 54.1. *Framework: Formal Methods in Conformance Testing.* February 1995.

[11] J. Kella. Sequential machine identification. *IEEE Transactions on Computers*, 20(3), March 1971.

[12] A.F. Petrenko and N. Yevtushenko. Fault detection in embedded components. In *IFIP 10th Int. Workshop on Testing of Communicating Systems*, Cheju Island, Korea, September 1997.

[13] A.F. Petrenko, N. Yevtushenko, and G.v. Bochmann. Fault models for testing in context. In *Proc. IFIP FORTE IX/PSTV XVI*, Kaiserslautern, Germany, October 1996.

[14] A.F. Petrenko, N. Yevtushenko, G.v. Bochmann, and R. Dssouli. Testing in context: framework and test derivation. *Computer Communications*, 19:1236–1249, 1996.

[15] D.P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, April

1989.

[16] S.T. Vuong and K.C. Ko. A novel approach to protocol test sequence generation. In *Proc. GLOBECOM'90*, December 1990.

[17] M. Yao, A. Petrenko, and G.v. Bochmann. A structural analysis approach to the evaluation of fault coverage for protocol conformance testing. In D. Hogrefe and S. Leue, editors, *FORTE'94*, 1994.

[18] J. Zhu and S.T. Chanson. Toward evaluating fault coverage of protocol test sequences. In *Proc. IFIP 14th Int. Symp. on Protocol Specification, Testing, and Verification*, Vancouver, Canada, June 1994.

10 BIOGRAPHY

Mr. Jinsong Zhu received his B.Sc and M.Sc in Computer Science from Tsinghua University, China in 1985 and 1987 respectively. He was a faculty member at Tsinghua University from 1987 to 1991. From 1995 to 1996, he was a Senior Software Engineer at the Infonet Software Solutions, Inc., Canada. He is currently working towards his Ph.D. degree at the Computer Science Department of the University of British Columbia. His research interests include protocol testing theory, high speed networks, mobile computing, and distributed object computing technology.

Dr. Son T. Vuong received his Ph.D. in computer science from the University of Waterloo, Canada. He had some industrial experience on image processing at CCRS and design of packet-switched networks at BNR (Nortel). He joined the faculty of the Department of Computer Science at the University of Waterloo for two years before moving to the University of British Columbia in 1983, where he was a founder of the Distributed Systems Research Group and is presently an Associate Professor. Dr. Vuong's areas of research interest include protocol engineering, distributed multimedia systems, high speed networks and mobile computing. Dr. Vuong served on several conference program committees and was the (co)chair and local organizer of numerous international conferences: ICDCS'95, PSTV'94, FORTE'89, IWPTS'88.

Dr. Samuel Chanson received his Ph.D. degree from the University of California, Berkeley in 1975. He was a faculty member at Purdue University for two years before joining the University of British Columbia where he became a full professor and director of its Distributed Systems Research Group. In 1993 he joined the Hong Kong University of Science & Technology as Professor and Associate Head of the Computer Science Department. He has chaired and served on the program committees of many international conferences on distributed systems and computer communications, including IEEE ICDCS, IFIP PSTV and IWTCS.

Dr. Chanson's research interests include protocols, high speed networks, multimedia communication and load balancing in workstation clusters. He has published more than 120 technical papers in the above areas.