COVERAGE ANALYSIS FOR EMBEDDED TESTING AND AN APPLICATION

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

Abstract In this paper, we present two new results on the properties of the coverage measure for embedded testing as proposed in [10], and an application of the measure to a practical protocol. First, we identify and prove the sufficient and necessary condition for loops to exist between the test context and the test tree being constructed. This allows us to avoid producing infinite test trees during test tree construction. Then we prove that the calculation of the coverage measure is NP-complete, which indicates that there unlikely exist efficient algorithms to compute the coverage in general. Finally, in order to demonstrate the concept and practicality of the coverage measure, we developed a tool set and applied it to a practical protocol extracted from the Universal Personal Computing system.

Keywords: Protocol testing, embedded test, fault coverage, universal personal computing

*Now with Abatis Systems Corporation, Burnaby, Canada.

1. INTRODUCTION

In this study, we focus on a type of protocol testing known as conformance testing [7], which is essentially a black-box testing strategy applied to protocol systems. In conformance testing, we try to determine if an IUT complies with its specification. This will enhance our confidence on interoperability between different implementations of a common specification. Conformance testing can also be used for certifying a product.

The research on protocol conformance testing has been targeting on both single (isolated) machine testing and embedded testing. In single machine testing, the machine offers full access of its interface and therefore thorough testing may be performed. On the other hand, in embedded testing the machine is embedded within a complex system and a tester can only indirectly test the machine via other surrounding machines known as the *test context.* The area of embedded testing is still relatively young, but interests are starting to rise [6, 5, 3, 10]. While most authors have focused on generating test suites, we are more interested in test coverage assessment of a test suite and in incrementally generating test suites based on the coverage requirement.

In [10], we have proposed an approach that evaluates the coverage of embedded testing by machine identification. We enhance the approach by solving the "loop prevention" problem (see Section 3.) with a pre-identified sufficient and necessary condition, and giving a proof on the computational complexity of the coverage calculation. Furthermore, we try to model a real-life protocol as embedded testing and evaluate the test coverage of a test suite for the embedded machine.

2. COVERAGE MEASURE FOR EMBEDDED TESTING

In embedded testing, we deal with a module embedded within a complex system which is referred to as a *composite* system. Modules in a composite system communicate with each other and provide an overall system service to its users. To test an embedded module that does not offer direct access to its interface, a tester can only try to infer the conformance of the embedded module by testing the composite system. The embedded module under test (MUT) is only exercised via its test context. In our study, we describe each module in a composite system as a finite state machine (FSM), and the composite system as a system of communicating finite state machines (CFSMs).

In a previous work [10], we have defined the coverage of a test suite with respect to a specification as follows:

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:*

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

where T-conforming machines are the machines that pass the test suite T, and S-conforming machines are the T-conforming machines that conform to S. $Cov(S,T,n)$ is inversely proportional to the number of machines that pass the test suite but do not conform to the specification.

In order to better reflect the coverage values, we will use the logarithmic scale in the above definition, i.e.,

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

To calculate $Cov(S,T,n)$, we propose an approach that identifies the MUT based on the test suite applied to the composite system. We first compute test trees for the MUT from the system test suite, and then "collapse" these trees into FSMs. The algorithm for generating the test trees are given in [10]. The algorithm gives us the set of T-conforming machines. The calculation of S-conforming machines is done at the composite system level, according to the following *conformance in context* (denoted $\mathbf{conf_c}$) relation:

$$M \,\mathbf{conf_c}\, S \text{ iff } C \circ M \,\mathbf{conf}\, C \circ S.$$

where **conf** is the conformance relation as used in single machine testing. This conformance relation takes into account the limited observability introduced by the test context. Based on this definition, the set of S-conforming machines can be produced.

3. LOOP PREVENTION

During construction of the test trees (TTs) for the MUT, a loop may occur where the context and the test tree keep exchanging internal events without producing an external output. This would create an infinite test tree and the TT construction algorithm would not terminate.

We propose an approach that pre-identifies the conditions under which a loop can happen, and then employs some measures to prevent it from happening during TT construction. An examination of the problem reveals that if the context machine contains cycles consisting of only

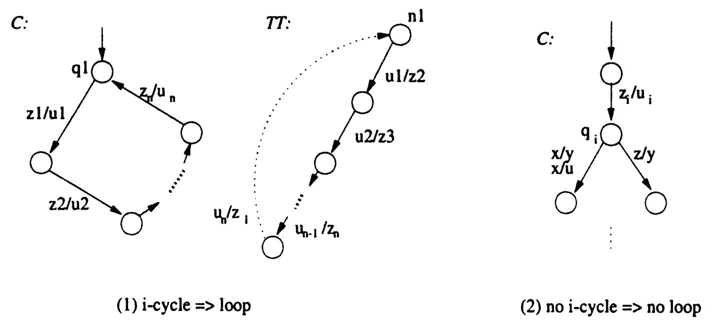

Figure 1.1 Proof for Theorem 1

internal events, it can cause a loop. A *cycle* is a transition path whose end state is the same as the start state. We call a cycle with all internal transitions an *internal cycle*, or i-cycle for short (similarly, i-path for internal paths). We prove the following theorem:

Theorem 1 *A loop between the test context and a test tree can happen if and only if there exists i-cycles in the test context.*

Proof: 1) i-cycles $\Rightarrow$ loop. Suppose there is an i-cycle in $(Z/U)^*$ in the test context C. Pick a state q_1 on the cycle, which is $z_1/u_1, z_2/u_2, ..., z_n/z_n$ (Figure 1.1). When C gets into q_1 via some external input, let the test tree TT be at node n_1. For C's output u_1, TT may select z_2 according to the TT construction algorithm, which causes C to output u_2. TT may select z_3 for u_2, causing C to output u_3. This may continue until C gets back to state q_1, and TT repeats the path from n_1. Therefore, there is a loop between C and TT.

2) no i-cycles $\Rightarrow$ no loop. Suppose there is a longest i-path in C, which is $\pi = z_1/u_1, z_2/u_2, ..., z_i/u_i$. Let C enter state q_i after z_i/u_i. The outgoing transitions from q_i must be of the form x/y, x/u, or z/y, since otherwise π would not be the longest i-path. For transitions of the form x/y and x/u, since x is an external input, it must be a test input from the test suite, and TT stays at the same node with no new edge introduced, *i.e.,* no loop will happen. For transitions of the form z/y, TT would have selected z, and y will be produced by C. This external output y is then compared against the test suite, and TT has a new edge u_i/z added. This continues until all test steps are exhausted at which point TT is terminated. Therefore no infinite loop will exist between C and TT. □

This theorem identifies the sufficient and necessary conditions for loops to exist. It is straightforward to identify all i-cycles. The next step is to find techniques to prevent loops once we recognize that they may happen.

Suppose there is an i-cycle, $\xi = z_1/u_1, z_2/u_2, ..., z_n/u_n$. We call a sequence $u_1/z_2, u_2/z_3, ..., u_{n-1}/z_n, u_n/z_1$ the complement of ξ and denote it as $\bar{\xi}$. It is the path in the test tree corresponding to ξ. In order to avoid loops, the test tree in construction must not contain the path $\bar{\xi}$. This can be done by recording the path in the test tree and whenever a new edge is added, it is compared against the complements of all i-cycles. The recording starts when the context enters into a state on an i-cycle. The test tree avoids the last step of an i-cycle complement by simply not choosing the output of that last step. This way the loop is broken and the test tree can be terminated in finite steps.

This technique is integrated into the TT construction algorithm to generate finite test trees. The trees are then fed to the tool COV to produce all T-conforming machines, which gives us the value of K_{TCM}.

4. NP-COMPLETENESS OF COVERAGE MEASURE CALCULATION

We already know from [8] that the calculation of the coverage measure $Cov(S, T, n)$ for single machine testing is NP-complete. We now ask the following question: Given a test suite T, and the test context C, is there a machine M with n states such that the composition of C and M agrees with T?

This question is important since during the construction of TCMs, we are essentially trying to find machines that accept the given test suite after composition with the test context. We claim that it is NP-complete and hence the coverage measure computation is also NP-complete. This result assures us that although embedded testing is in general more complicated than single machine testing, the computational complexity of its test coverage evaluation is not.

Theorem 2 *Let T be a test suite and C the test context. It is NP-complete to find an n-state machine M, which interacts with C via a predefined channel such that the composition of C and M accepts T.*

Proof: Let us call this problem ESAT. First we show that ESAT is in the domain of NP. A non-deterministic algorithm can guess a solution machine M for ESAT, and then check if the composition of C and M accepts T. Suppose the length of T is l, the checking can be done by applying the test steps one by one. For each test step, we only retain the stable global states which will be reached unless a livelock occurs. If a livelock is detected, the checking halts with the answer "no"; otherwise

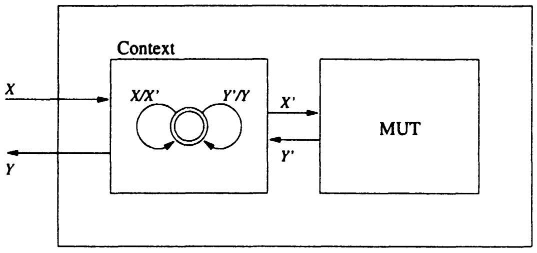

Figure 1.2 A special case of ESAT

the test step can be verified in $O(m)$ time where m is the number of internal exchanges between C and M. The whole test suite can then be checked in $O(lm_{max})$ where m_{max} is the maximal number of internal exchanges for all test steps. Therefore, the checking of whether M is indeed a solution can be performed in polynomial time.

Now we show that ESAT contains a known NP-complete problem as a special case. Consider a restricted case of ESAT: the context machine C is a one state FSM as in Figure 1.2, and the internal interactions between C and a MUT is X' and Y', which are simply a renaming of X and Y respectively. Under this restriction, C is essentially a "pass-through" module, and ESAT becomes the single machine identification problem which is known to be NP-complete [1]. □

The NP-completeness of the ESAT problem indicates that it is unlikely that there exists polynomial algorithm to compute our coverage measure. Therefore, heuristics may be the best approach to achieve good performance. On the other hand, protocols should be structured with smaller components, and large monolithic systems should be avoided.

5. APPLICATION

To validate the approach, an experimental tool set called ECOV has been implemented. We applied the tool set to a protocol extracted from a Universal Personal Computing (UPC) system [9, 4] which we developed to support personal mobility on the Internet which is independent of the user's location, motion, and platforms. This new computing paradigm retains one's personal computing environment (PCE) and access capabilities wherever one happens to be. Personal mobility is one step further than the traditional terminal mobility where the physical devices migrate.

UPC is modeled as a set of communicating objects distributed in the Internet, with some objects invisible to the outside world. Testing of these objects in an embedded manner presents a suitable application of our method in a multiple module setting.

5.1 UPC PROTOCOLS

The UPC system employs a number of protocols that support personal mobility on the Internet. This includes the middleware support that allows a mobile user to access the network wherever he roams, the mobile user location management protocol, and mobile user PCE management. We shall focus on the first protocol. The latter two protocols can be found in the references [4, 9].

The middleware support protocol describes the interactions between the user home agent (UHA), the terminal initial agent (IA), and the user terminal agent (UTA). This protocol offers a virtual global network service to the mobile user. Using the CFSM formalism, we model these objects as a system of communicating finite state machines. In order to make ~~the system more~~ understandable and more manageable, we abstract away some non-essential events, and take the liberty in choosing the representative values for the protocol data parameters. Furthermore, where non-determinism arises, we will determinize it appropriately. The abstracted protocol is described as two communicating FSMs in Figure 1.3, where the initial agent and the user terminal agent are combined since the two mostly communicate between themselves. This protocol can be described as a two-CFSM system.

For demonstrating our embedded testing strategy, we consider the case where the home agent is an embedded module. This corresponds to the situation where a mobile user roams to a foreign location, and wishes to access his personal PCE from there. The user has no direct access to the home agent; what he directly interacts with is the initial agent and the terminal agent. The testing issue is therefore how one can test the functionality of the home agent from the viewpoint of a mobile user.

We have composed the initial agent, the terminal agent, and the home agent to produce a composite system (also shown in Figure 1.3). This global system models the system behavior that a tester can observe and control.

5.2 UPC TEST SUITE EVALUATION

Test suites for embedded modules can be generated using methods such as that presented in [5]. However since no tool is available for the method, we chose to evaluate a test suite directly generated from the composite system. We use the UIO-method for this purpose, in order

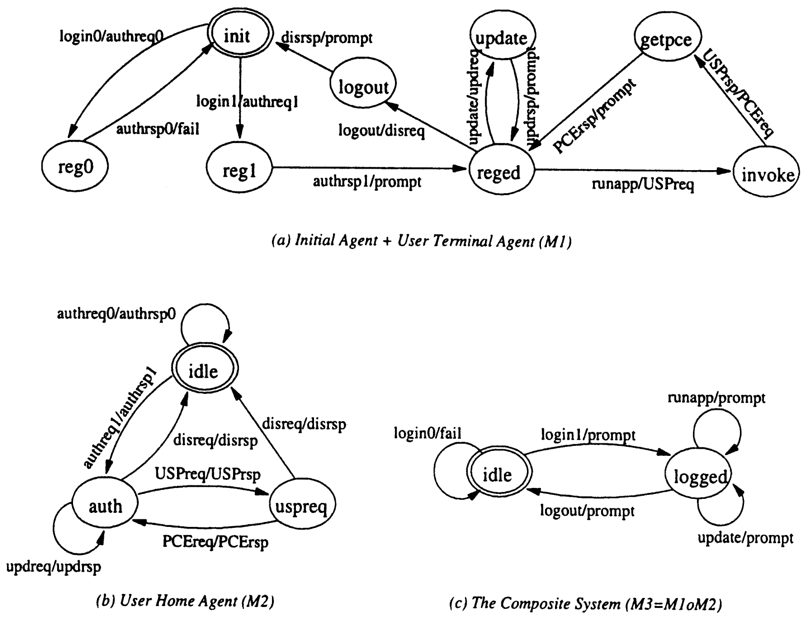

Figure 1.3 UPC middleware protocol

to get an idea on how the method fares for embedded module testing. The composite machine is completed first, with the assumption that for unspecified transitions, the machine outputs a null output and stays in the same state. It is easy to verify that *login1/prompt* is a UIO sequence for the *idle* state, and *logout/prompt* is a UIO sequence for the *logged* state. This leads to the following test suite:

```
TS = {
  r/- login0/fail login1/prompt
  r/- runapp/null login1/prompt
  r/- update/null login1/prompt
  r/- logout/null login1/prompt
  r/- login1/prompt login0/null logout/prompt
  r/- login1/prompt login1/null logout/prompt
  r/- login1/prompt runapp/prompt logout/prompt
  r/- login1/prompt update/prompt logout/prompt
  r/- login1/prompt logout/prompt login1/prompt
}
```

This test suite has complete fault coverage for the composite system, as can be verified with our tool COV. The question remains how well it does in testing the UHA as an embedded module. Feeding this to our tool box ECOV, one test tree was produced. It represents a test suite for the UHA as follows:

```
TS = {
  r/- aureq0/aursp0 aureq1/aursp1
  r/- aureq1/aursp1 USPreq/USPrsp PCEreq/PCErsp disreq/disrsp
  r/- aureq1/aursp1 updreq/updrsp disreq/disrsp
  r/- aureq1/aursp1 disreq/disrsp aureq1/aursp1
}
```

This tree is then collapsed using the tool COV, and the resulting FSMs are composed with the context machine IA/UTA to determine the conformance in context. We did some experiments regarding the upper bound n on number of states for the solution machines (mutants of the UHA machine). For $n = 3$, there were 4031 solutions, 5 of which are conforming, therefore we have $K_{TCM} = 4031$, $K_{SCM} = 5$, and $Cov(S, T, 3) = 1/(\lg 4031 - \lg 5 + 1) = 25.6\%$. For $n = 2$, there were 196 solutions, with 4 conforming, which leads to $Cov(S, T, 2) = 1/(\lg 196 - \lg 4 + 1) = 37.2\%$. In the extreme case where $n = 1$, we get a unique conforming solution with only one state, ie., $Cov(S, T, 1) = 100\%$. This indicates that the lower the upper bound n, the higher the fault coverage.

The result shows that it is generally not enough to test an embedded module with a test suite complete for the composite system. Additional test cases are needed to detect potential faulty machines. This can be done by incrementally generating test cases that distinguishes the potential faulty machines based on test coverage requirement.

6. CONCLUSIONS

We have presented our work on some new results on coverage measure for embedded testing, and on the application of the measure to a practical protocol. The work offers a better way to avoid infinite test trees and a better understanding of the computational complexity for the coverage measure. The application presents a practical situation where embedded testing can be applied, and shows that in general we cannot simply take a test suite generated for the composite system for embedded testing, as the fault coverage may be weakened depending on the upper bound of number of states in the mutants.

Our coverage measure is essentially based on machine identification which offers an accurate fault coverage for a test suite. It allows incremental test suite generation based on coverage requirement and facilitates fault diagnosis with faulty suspects explicitly constructed. The price for the advantages is higher computational cost, which can be eased by developing better heuristics and more structured, decomposable protocols.

The recent development of embedded systems on the Internet has captured significant attentions [2]. It is practically important to see if these embedded systems can be tested with an embedded testing architecture. Internet provides a wide range of testbeds for distributed computing with many types of embedded systems, such as routers, Internet phones, NetTVs, WebTV set-top boxes, and so on. Modeling these systems (with their environments) as communicating finite machines and applying our testing methodology would contribute a lot to the reliability of the systems.

References

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

[2] IEEE Computer Society. *IEEE Internet Computing.* July 1998.

[3] 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.

[4] Y. Li and V. Leung. Supporting personal mobility for nomadic computing over Internet. *ACM Mobile Computing and Communications Review*, 1(1):22–31, April 1997.

[5] 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.

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

[7] D. Rayner. OSI conformance testing. *Computer Networks and ISDN Systems*, 14(1), 1987.

[8] 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.

[9] J. Zhu, M. Toro, V. Leung, and S. Vuong. Supporting universal personal computing on the Internet with Java and CORBA. *Concurrency: Practice and Experience*, 10(1), 1998.

[10] J. Zhu, S.T. Vuong, and S.T. Chanson. Evaluation of test coverage for embedded system testing. In *Proc. IWTCS'98*, Tomsk, Russia, September 1998.