[1]An Approach for Concurrent FSM-based Test Case Generation

Ana Maria Ambrosio[1]

[1]Ground Systems Development Division (DSS)
ana@dss.inpe.br

Eliane Martins[3]

[3]Institute of Computing (IC)
State University of Campinas (UNICAMP)
eliane@ic.unicamp.br

Solon V. de Carvalho[2]
Nandamudi L. Vijaykumar[2]

[2]Associated Laboratory of Computing and Applied Mathematics (LAC)
National Institute for Space Research (INPE)
{solon, vijay}@lac.inpe.br

Abstract

This paper presents an approach for black-box test case derivation from a set of concurrent FSM, in which the product machine is not generated. The approach is based on the concept of independent and communicating transitions. An algorithm to recognize the communicating transitions from the concurrent-FSM-based specification is presented. A set of test cases was generated supported by an existing tool able to generate tests for simple FSM-based specification, the Condado. The test case suite generated according to the approach was then compared with the test suite generated from the product machine, also using the Condado. In order to evaluate the effectiveness of the test suite generated by the proposed approach, a set of automatically generated mutants was used. The code-based interface mutants were used as a fault model to support the comparison between both the test case sets. A simple example illustrates the approach and a comparison is made to an empirical study. Preliminary results pointed out simplicity and effectiveness of the approach over the fault model in the empirical evaluation.

1. Introduction

Space agencies, such as the National Institute for Space Research are nowadays more involved in acquiring third party software products developed by industrial and commercial companies. These products should be integrated with others and have to be reliable. Then it is important to develop technologies to test such products during the acceptance phase of the delivery software. The contractors make the software specification as part of contract before the development starts and have to ascertain whether the delivered implementation really conforms to the respective specification.

One important kind of verification to support the acceptance process is the conformance testing, as defined for ISO protocol testing [5]. One-way to check the conformance of the specification against the implementation is to apply a test suite and compare actual against expected results. In this case, the test case suite should not be designed on code-basis, as it is not available, instead, it should be based on the specification requirements, which indicates how the software should behave.

This effort, known as conformance testing, has been strongly explored in protocol area [1], [3], [6], [7], etc. The difficulties in conformance testing are related to insufficient techniques and supported tools to test design [11].

Some kind of concurrent systems of the space area, like a satellite simulator, which has several components, each one having its own behavior, may be specified as a set of Finite State Machines (FSMs). So the test techniques applied to protocols may be also applied to other applications whose specification can be given as a set of FSMs. Each FSM specification represents

[1] Workshop de Teses e Dissertações da CAP/INPE- novembro de 2003 – São José dos Campos-SP- Brasil

the behavior of a different parallel component. The global state of such a system is, then, given by the product of the FSMs.

In the literature, specification with a set of FSMs is generally found as Communicating FSM (CFSM). To generate test cases from a CFSM-based specification one may not count on the conventional approaches of automatic test case generation, in which the whole system behavior, given by the product of the machines, has a considerable size.

Techniques to avoid the state explosion in the CFSM-based specification test cases generation may be found in [4], [8], etc.

We propose an approach to systematically derive test cases from a set of FSMs designed as Meale machines [10], where the communication transitions are not limited to the rendezvous of CFSM specification. The rendezvous communication means the synchronized message exchange between two processes [7].

The approach suggests to incrementally generate test cases based on the classification of the transitions, viz., independent and communicating. First, test cases related to each component are separately generated. In this phase, the test cases based on the independent transitions are created. Later, the communicating transitions are identified from the set of FSMs. Then, considering only the communicating transitions, a new FSM is created. Each FSM undergoes through an existing tool named ConDado [9], which is used to derive the test cases. Details of the approach are presented in Section 2. The method of how the test cases are generated by the ConDado is described in Section 3.

Having the test suite generated according to the proposed approach, the question now is: *what is the effectiveness of this test suite?* In order to answer this question we have performed an experimental evaluation of the coverage of the test suite against the set of code-based interface mutants. The resulted mutation score was compared against the results of the mutation applied to a test suite created from the product machine of the set of the specified FSMs.

The specification that illustrates this experiment has 3 components. The behavior of each component is modeled by a FSM comprising 3, 2 and 2 states. The system behavior, given by the product machine has 12 states and 22 transitions.

The mutants were automatically generated by the tool named ProteumIM [2] whose characteristics are given in section 3.

Section 4, summarizes the experimental evaluation and section 5 concludes the paper pointing out future directions.

2. The Approach

Model

We assume that the specification of a concurrent system be given in a set of FSMs, where each FSM represents an orthogonal component of the system. Each FSM comprises a set of states connected by transitions. Each transition comprises an event (input) and an action (output). *Events* may be external (explicitly stimulated) or internal (automatically stimulated by an output of any transition); an *action* may cause an output or trigger another event. A *transition* links a source-state to a target-state, and it is represented as: *t = (source-state, target-state, event [condition], actions).*

Figure 1 illustrates a specification with three orthogonal components, M_1, MB and M_2. M_1 comprises three states: W_1, P_1 and F_1. In the transition from the state W_1, to P_1, there is a condition *[not inB1]* associated to the event *a*, which means the transition will be triggered only if the component MB is not in the state B1. And, in the transition from W_1 to P_1 the action *i* will cause a transition in the component MB.

The proposed method to generate specification-based test cases is founded in the definition of communicating and independent transitions, whose definitions are given in the following.

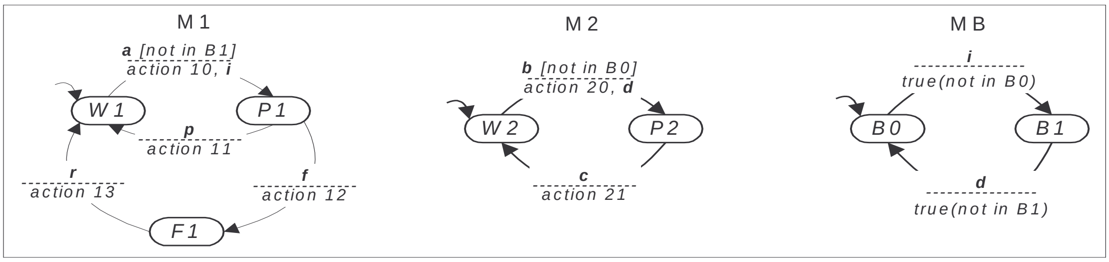

Figure 1 – Specification example

Communicating and independent transitions

A communicating transition is a transition from one state to another in the same FSM that comprises one of the following features:

- broadcast event – event that appears in more than one machine,
- event caused by an action specified in another machine,
- condition associated to state – condition that indicates the transition will be fired only if another machine is in such a specified state.

In Figure 1, four communicating transitions are illustrated, which are:

(W_1, P_1, *a[not in B1]; action10*, *i*),

(W_2, P_2, *b[not in B0]*, *action20*, *d*),

(B1, B0, *d*, *true(not in B1)*),

(B0, B1, i, *true(not in B0)*).

The others are independent transitions.

Method

In short, the method to generate test cases from a set of FSMs-based specification consists of the following steps:

(i) Generate a set of test cases to each FSM, separately;

(ii) identify the *communicating transitions* among all the FSM;

(iii) create a FSM, named *communicating machine*, comprising all the communicating transitions;

(iv) generate test cases to the communicating machine;

(v) apply, on the implementation, the test sequence containing the test cases generated in both steps (i) and (iv).

Communicating machine

The algorithm to generate the communicating machine is the following:

1. To each machine M_k do:
   - Create a transition set TS(k), where each element is a transition t = *(source-state, target-state, event, [condition], actions)*.
   - Identify the initial state, q_0 of M_k
2. Create a set SS(i) with the *communicating transitions*, to each machine:
   - To each TS(i):

     /* identify broadcast events */
     - Include the transitions t_i to SS(i) where $t_i.event \in$ TS(i) $= t_j.event \in$TS(j);

     /* identify state-dependable transitions */
     - Include the transitions t_i in SS(i) where $t_i.cond$ depends on a state $\underline{s_j}$ and Mj ≠ Mi
     - Insert transitions t_j of Mj in SS(j), which inputs or outputs the state $\underline{s_j}$ identified in previous step

     /* communicating transitions */
     - Insert the transitions of Mi and the transitions of Mj in SS(i) and SS(j), where $t_i.action = t_j.action$ and Mj ≠ Mi

     /* make each machine SS initially connected
     - insert in SS(i) the transitions that:
       - take any state in SS(i) to the initial state.
       - allow the states of SS(i) to return, by any path, to the initial state.
       - allow the machine in SS(i) be connected.
3. Create the communicating machine as a product of the machines in all SS(i).

3. Testing tools: ConDado and ProteumIM

For the experimental evaluation two tools in their prototype versions were used. These tools were developed in the context of academia researches: ConDado of the University of Campinas (IC/UNICAMP) and ProteumIM of the University of São Paulo (ICMC/USP). Their general description is given below.

ConDado

ConData [9] is used to create the behavioral test cases from a FSM-specification. This tool implements the transition-tour method for graph tour, with depth-first search. Each path, which comprises a set of transitions from the initial state to the initial state, is a test case.

The FSM specification should be written in a private protocol specification language to be interpreted by the tool.

The criterion adopted for deriving the test suite is to cover all-path with one-loop. With this ambitious criterion, the suite comprises a large number of test cases, as it may be seen in Table 1.

ProteumIM

The ProteumIM tool [2] has a set of mutation operators, which are automatically inserted in a C program generating the mutations from the original program. The mutation operators are only code-based covering interface errors for C programs. Besides generating the mutants, the tool also supports the test execution of a set of test cases (by test cases, in this context, we mean inputs to the programs). Each test case is submitted over every created mutant. Among the analysis information obtained from the test execution, the ProteumIM provides the user with the number of mutants that are alive and equivalent. Mutants that are alive are those programs in which the inserted error was not detected by any of the test cases. The equivalent mutants are those mutants that did not modify the logical flow of the program. Any mutant set as equivalent is not considered in the mutation score. The mutation score is calculated by the expression presented in figure 3:

$$\boxed{\frac{dead\ mut.}{total - equivalent\ mut.}}$$

Figure 3: Mutation score

4. Experimental evaluation

In order to evaluate the goodness of the test suite, we performed an experiment consisting of the following steps:

(i) implement, in C, the specification given in Figure 1;

(ii) generate code-interface-based mutants using ProteumIM,

(iii) generate the specification-based test suite named TS_{NEW}, according to the new approach: each machine Mi and the SS machine were individually undergone to ConDado. The M_is are the M1 M2 and MB, as illustrated in Figure 1. The SS machine generated for this example is illustrated in Figure 2. Each Condado run generated a small set of test cases comprising 2, 1, 1 and 18 test cases respectively. TS_{NEW} is the union of each small set of test cases consisting of 22 test cases;

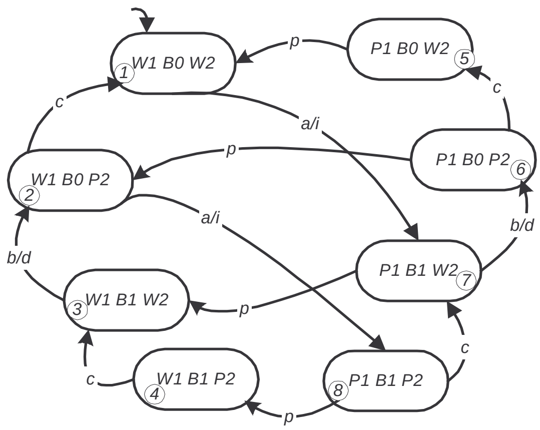

Figure 2. SS machine

(iv) generate the product machine from the components M1 M2 and MB. PerformCharts [12] tool may generate this product automatically, but in this example, the product was done manually. The product machine comprises 12 states and 22 transitions. Each state is named with the mnemonic of the state it represents. The resulted FSM is illustrated in Figure 3;

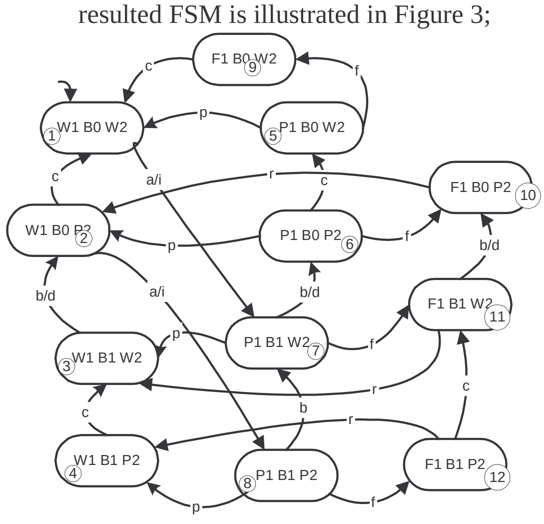

Figure 3. Product machine

(v) generate the test suite named TS_{PM}, by undergoing the product machine specification to ConDado. For the given FSM product, 766 test cases were generated;

(vi) apply both test suites, TS_{PM} and TS_{NEW} to the programs (original and mutants) generated by ProteumIM;

(vii) analyze the interface code-based fault coverage of both test suites. These results is presented following in this section.

Experimental Results

The resulting figures of the experiment described above are summarized in the tables.

Table 1 shows the figures related to the test suite sizes. The columns represent each of the test suites TS_{PM} and TS_{NEW}. The first line states the number of test cases generated by ConDado for each set. The second line shows the number of test cases that effectively killed any mutant. The difference shows that a small number of test cases are enough to find errors caused by mistakes on interface among coded functions.

Table1: Number of test cases

Test cases:	TS_{NEW}	TS_{CM}
Generated	22	766
Effective	7	6

Table 2 gives the total mutants. It is worth observing that the number of mutants depends only of the source code (C program).

Table2: Mutants Number

Number of Mutants:	
Total	1019

Table 3 and 4 show the fault coverage for two situations respectively: (i) before setting equivalent mutants and (ii) after setting them.

Table3: Fault coverage –all mutants

Mutants:	TS_{NEW}	TS_{CM}
Alive	143	149
Equivalent	0	0
Score	0.8597	0.8538

Table 4: Fault coverage –without equivalents

Mutants:	TS_{NEW}	TS_{CM}
Alive	7	17
Equivalent	137	132
Score	0.9932	0.9808

Equivalent mutants, in this version of ProteumIM should be manually marked. In setting them, the score was considerably augmented. For the TS_{NEW} set of test cases the score reached 0.99 as illustrated in Table 4. In this case, the score would be 1 if the system were strongly specified, as the mutants that are alive correspond to errors in the parts of code dealing with non-specified *events*. For example, the code treats the situation when none of the events: *a, p, r, f, b, c, i* and *d* is an input to the program and the specification says nothing about any different event. For a correct conformance, the code should have no treatment to them. However, some code has already a structure to treat non-specified events as the command switch of C. The default is part of the command forcing the programmer to include at least a message of error in the code.

5. Conclusion and Future Work

The work presents an approach for black-box test derivation and an experimental evaluation of the generated test cases. Every test case was derived from simple FSM-based specification by the ConDado tool. The approach is an alternative to avoid generating the product machine from a set of FSMs-based specification. In this way, the number of automatically generated test cases is also reduced. The approach is yet helpful and simple for concurrent applications.

Although part of the approach was applied manually, the experiment was invaluable as it mimics a real world situation when the contractor has to validate a system, when the code is not available.

The work also showed a kind of measuring for conformance testing: the test cases derived from a high level specification were checked against code-based interface faults. All the mutation was applied and automatically analyzed with the support of the ProteumIM tool.

The results pointed out that the proposed approach seems to be useful and feasible. In the case of many orthogonal machines, different communicating machines may be combined for test purposes.

Future works shall be carried out on combining test cases generated for conformance test, obeying the standard for protocol testing, the ISO-9646, and the fault cases provided by the fault injection techniques for validating the robustness of a software system.

References

[1] Bochmann, G.; Petrenko, A . – Protocol Testing: Review of Methods and Relevance for Software Testing – Proceedings of the 1994 International Symposium on Software Testing and Analysis, p.109-124, August 17-19, 1994 – Seattle, Washington, USA.

[2] Delamaro, M.E. Mutação de Interface: Critério de Adequação Interprocedural para o Teste de Integração – Ph D. Thesis – Instituto de Física de São Carlos - Universidade de São Paulo, 1997.

[3] Dssouli, H.; Salek, K.; Aboulhamid, E; En-Nouaary, A ; Bourhfir, C. Test Development for Communication Protocols: Towards Automation. Computer Networks 31, 1999, 1835-1872.

[4] Henninger, O. On test case generation from asynchronously communicating state machines. 10th International Workshop on Testing of Communicating Systems, September, 1997.

[5] International Organization for Standardization/International Electrotechnical Commission – "Conformance Testing Methodology and Framework", International Standard IS-9646. ISO, Geneve, 1991. Também: CCITT X.290-X.296.

[6] Lai, R. A survey of communication protocol testing. The Journal of Systems and S/w, 62, 2002, pp. 21-46.

[7] Lee, D.; Yannakakis, M. Principles and Methods of Testing Finite State Machines – a Survey – Proceedings IEEE, 84 (8): 1090-1123. 1996.

[8] Li,J.J.; Wong,W.E. Automatic test Generation from Communicating EFSM (CEFSM)-based Models. Proceedings of the 5th IEEE Intnl Symposium on OO Real-Time Distributed Computing, 2002.

[9] Martins,E; Sabião,S.B; Ambrosio, A.M. ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems. S/w Quality Journal, 8 (4) (1999) 303-319.

[10] Menezes, P.B. – Linguagens formais e Autômatos – 2001 - 4a. edição - número 3 -Ed. Sagra Luzzatto.

[11] Tretmans, J.; Belinfante, A. Automatic Testing with Formal Methods. In Proceedings of the Conference on Software Testing, Analysis and Review. EuroSTAR´99, November, 1999.

[12] Vijaykumar, N. L.; Carvalho, S. V.; Abdurahiman, V. On proposing Statecharts to specify Performance Models. International Transactions in Operational Research, 9(3), (2002) 321-336.