Proceedings of the 10th WSEAS International Conference on SYSTEMS, Vouliagmeni, Athens, Greece, July 10-12, 2006 (pp50-55)

Testing Discrete-Event Simulation Programs Using Black-Box Techniques

EVANGELOS KEHRIS
Department of Business Administration
Technological Educational Institute (T.E.I.) of Serres
Terma Magnisias, 621 24 Serres
GREECE
kehris@teiser.gr

Abstract: - Simulation program development is supported by simulation languages and simulation-specific libraries developed in general-purpose programming languages. Although these software tools are widely discussed in the simulation literature, simulation programming testing, an important phase of a simulation study which aims to uncover errors in the simulation program, is to a large extend under-studied. This paper discusses two complementary testing techniques that have been found helpful in testing simulation programs developed using discrete-event simulation-specific libraries. The testing techniques are presented and adopted for use in simulation, modifications need to be introduced to the simulation library in order to apply the techniques are discussed and software tools that automate the testing process are presented. The power of the techniques to identify errors imputed in a simulation program is demonstrated.

Key-Words: - Simulation, Testing, Equivalence Partitioning, X-Machines

1 Introduction

Simulation is often used for the study of real-life complex systems. In a typical simulation study, initially, a conceptual model of the system is constructed and based on this conceptual model, the simulation program is developed. Data driven simulators (such as described in [1], [2], [3], [4]) facilitate the development of the conceptual model by representing the simulated system as data and automate the development of the corresponding simulation program. Thus, in the cases that a simulator may capture the characteristics of the system under study with the necessary level of accuracy, simulation program testing is not necessary since it is legitimately assumed that the program generation capability of the simulator has been exhaustively tested by the simulator developer and may be trusted by the user. However, simulation experts quite often have to manually develop simulation programs since existing simulators do not always provide the required functionality. In these cases, the simulation program is developed in a simulation language or in a general-purpose language using simulation-specific libraries (e.g. [5], [6], [7]). The testing of the simulation programs thus developed is a necessary step in order to establish that the conceptual model has been properly transformed to program.

The simulation languages and the simulation-specific libraries that have been developed are well documented and widely publicized in academic journals and international conferences. The testing of the simulation programs developed using these languages and simulation libraries has not received the corresponding attention. To begin with, the dominant trend in simulation literature is to discuss testing together with validation and verification. Thus, nearly all methodological simulation papers recommend validation verification and testing (VV & T) as necessary activities that need to take place throughout the simulation development life cycle, they provide a more or less widely acceptable categorization of VV & T techniques (usually without distinguish between validation techniques, verification techniques and testing techniques) and conclude by briefly describing existing VV & T techniques. In short, it could be stated that the majority of the simulation papers that deal with testing remain at a general description of existing testing techniques without describing in depth how to employ these techniques for the testing simulation programs. As a result, simulation-specific testing issues such as: the implementation of existing testing approaches to simulation or the modification of the simulation libraries in order to support the testing of the simulation program are not discussed in the simulation literature.

This paper discusses two complementary testing techniques that have been found helpful in testing simulation programs developed using discrete-event simulation-specific libraries. The testing techniques are described and adopted for use in simulation, modifications need to be introduced to the simulation library in order to apply the techniques are discussed and software tools that automate the testing process are presented. Finally, the power of the techniques to identify errors imputed in a simulation program is demonstrated.

According to Myers [8] testing is the process of executing a program with the intent of finding errors. Effective testing requires careful selection of specific sets of data so that the functionality of the program is satisfactorily tested. The data used for testing purposes are called test-cases and for each test-case, the expected output must also be determined. Additional code may be necessary to be added to the program under test in order to collect information about program behaviour during execution. The testing of the program is then conducted by comparing the output produced by the program under test to the expected output: any difference between the simulation-generated output and the expected output identifies an error in the simulation program.

Black-box testing techniques [8] base the generation of test-cased on the specification of the program under test while white-box testing techniques examine the internal structure of the program under test in order to generate the test-cases. In this paper we study the employment of two black-box techniques for testing the program that simulates the system described in the section 2.

2 The simulated system

Consider a simple manufacturing facility which manufactures product parts. Each product part is uniquely identified by an identification number. The manufacturing facility consists of buffers which are storage spaces of limited capacity and simple machines which carry out the manufacturing processing. Parts may be added in a buffer only if there is available space in it, while the parts removed from a buffer depend on the buffer discipline.

The parts that are required to be processed by a machine are placed in a buffer which is called input buffer, while the parts that have been processed by the machine are stored in another buffer called output buffer.

When the machine is idle and there are parts stored in the input buffer, the machine may start the processing of a part: The first part p placed in the input buffer is removed from the input buffer and the machine starts processing it. Thus, the FIFO discipline for input buffer is used. The processing of the part lasts for t time units. If at the completion of the part processing, the output buffer is not full, then part p is placed in the output buffer and the machines either becomes idle or starts processing another part depending on whether the input buffer is empty or not. If, however, the output buffer is full when the machine completes the processing of a part, the part p may not be removed from the machine and thus then machine is blocked. The machine is unblocked when space becomes available in the output buffer. It is assumed that the machine does not require setup and does not fail.

3 Simulation system specification

This section describes the specification of the simulation system described previously. The specification of the system is based on the X-Machine formalism [9] and constitutes the basis upon which the black-box testing techniques are developed.

3.1 Buffer specification

The buffer memory is: BM = (PARTS $\times$ Capacity) where PARTS represents the set of all the parts contained in the buffer in a given moment and Capacity is the maximum number of parts that may be stored in the buffer.

Two functions are necessary to describe the behaviour of the buffer: add_part is a function responsible for adding a new part into the buffer and remove_part is a function which removes a part stored in the buffer. We will use the following notation for the specification of a function:

f (in, mem) = (out, new_mem) guard

This notation is read as: function f accepts as input in and operates on memory mem; if guard is satisfied then function f changes the memory into new_mem and produces the output out.

The specification of add_part and remove_part is given next using the above notation:

add_part (p, (Parts, Capacity)) =
(part_added, (Parts $\cup$ p, Capacity)
if p $\notin$ Parts $\wedge$ card (Parts) $\leq$ Capacity – 1

remove_part (p, (Parts, Capacity)) =
(part_removed, ((part – p), Capacity))
if p $\in$ Parts $\wedge$ card (Parts) $\geq$ 1

Where:

- p is the part that is going to be added in the buffer by add_part
- (Parts, Capacity) is the memory of the buffer as stated earlier
- part_added is the output produced by the function add_part when it is executed
- card (Parts) is the cardinality of the set Parts.

To complete the specification of the buffer, we develop its X-Machine description. All the elements of this X-Machine have been described earlier, so it only remains to develop the associated finite state automaton of the X-Machine which is shown in Figure 1.

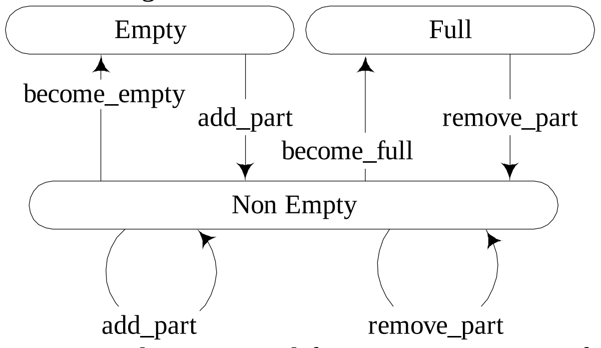

Figure 1: The associated finite state automaton for the Buffer. Initial state: Empty.

The functions add_part and remove_part that have been defined earlier, here have been renamed. Thus, the function becomeFull is the function add_part, while the function become_empty is the function remove_part.

3.2. Machine specification

The machine memory is:
MM = (Time$\times$ IN $\times$ PROC $\times$ OUT $\times$ BTime $\times$ DUR) where

- Time is the current simulation time
- IN is the set of parts contained in the machine's input buffer
- PROC is the set of parts currently being processed by the machine
- OUT is the set of parts currently stored in the machine's output buffer
- BTime is the simulation time the machine is expected to complete its current operation
- DUR is the duration of the operation

The behavior of the machine is described by five functions: start and end_process (which model the commencement and the completion of a machine operation respectively), reset (which models the fact that a machine just finished an operation may proceed with another operation) and block and unblock. The definitions of the function start follows, using the notation explained earlier. The other functions are defined similarly.

start (check_start, (now, in, nil, out, nil, dur)) =
(proc_starts, (now, (in-p), p, out, now+dur, dur),)
if (in $\neq$ empty $\wedge$ Proc = nil $\wedge$ BTime = nill)

The associated finite state automaton of the X-Machine that corresponds to the machine is shown in Figure 2.

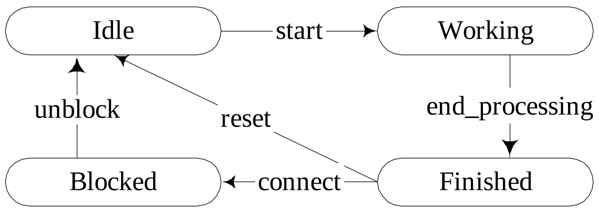

Figure 2: The associated finite state automaton for the Machine. Initial state: Idle.

Having completed the specification of the Buffer and the Machine it is now possible to generate the test-cases that will be used for testing their implementation.

4 Test-case generation

In this section we present how two black-box testing techniques may be used for the testing of simulation entities. The first black-box technique (equivalence partitioning) is used to test the individual functions of a simulation program, while the second technique (X-Machine testing) is used to test the integration of functions.

4.1 Test-case generation based on equivalence partitioning

In equivalence partitioning equivalence classes of test data are generated: valid equivalence classes representing valid values to variables while invalid equivalence classes representing erroneous variable values. The equivalence classes are identified based on the specification of the functions.

Lets consider the guard of the function add_part of the Buffer. Two conditions may be identified which satisfy the guard: according to the first condition a part p that is not hold in buffer ($p \notin$ parts) is added in a buffer having more than one empty spaces (card (parts) < capacity – 1). A test-case that represents this condition is to add part p2 to a buffer with capacity 5 that holds part p1. This test-case is described as: MB = (<p1>, 5), p = p1. According to the second condition a part p that is not hold in buffer ($p \notin$ parts) is added in a buffer having exactly one empty spaces (card (parts) < capacity – 1). A test-case that represents this testing condition is to add part p2 to a buffer with capacity 2 the holds part p1, i.e. the test-case: MB = (<p1>, 2), p = p2. These two input conditions constitute the valid equivalence classes for the Buffer function add_part. Invalid equivalence classes need also be derived. The valid and invalid equivalence classes for the functions of the Buffer, together with their corresponding test-cases are shown in the Table 1.

BUFFER
Function: add_part

a) valid equivalence classes	
$p \notin$ parts $\wedge$ card (parts) < capacity – 1	1. MB = (<p1>, 5), p = p2
$p \notin$ parts $\wedge$ card (parts) = capacity – 1	2. MB = (<p1>, 2) p = p2
b) invalid equivalence classes	
$p \in$ parts $\wedge$ card (parts) < capacity – 1	3. MB = (<p1>, 5), p = p1
$p \notin$ parts $\wedge$ card (parts) > capacity –1	4. MB = (<p1, p2>, 2), p = p3

Function: remove_part

a) valid equivalence classes	
$p \in$ parts $\wedge$ card (parts) > 1	5. MB = (<p1,p2>, 5), p = p2
$p \in$ parts $\wedge$ card (parts) = 1	6. MB = (<p1>, 5), p = p1
b) invalid equivalence classes	
$p \notin$ parts $\wedge$ card (parts) > 1	7. MB = (<p1, p2>, 5), p = p3

Table 1: Test-cases for the Buffer functions generated by the equivalence partitioning approach.

The functions that represent the Machine functionality have also been treated in a similar manner, in order to derive the appropriate test-cases. In total ten test-cases (three of them being valid and seven invalid equivalence classes) have been derived for the Machine.

4.2 Test-case generation based on X-Machine testing

The X-Machine test-case (XMTC) generation is an extension of Chow's W-method [10] and is presented in detail in [9]. XMTC generation requires the identification of two sets: the characterisation set and the state cover set. Informally, a characterisation set W is a set of input sequences for which any two distinct states of the machine are distinguishable. The state cover S is a set of input sequences such that all states are reachable by the initial state.

The implementation of the XMTC generation algorithm (for k = 1) generates 96 test-cases for the buffer and 108 for the machine. Due to space limitations, two short extracts of these test-case are shown next:

Test-case for the Buffer
B1. e:become_empty
B2. e:ignore_add
B3. e:add_part:become_empty
B4. add_part:become_full:ignore_add:become_empty
B5. add_part:become_full:ignore_add:ignore_add
B6. add_part:add_part:become_empty
B7. add_part:become_full:remove_part:add_part: become_empty
B8. add_part:become_full:remove_part:add_part:ignore_ add
B9. add_part::become_full::ignore_add::become_empty

Test-case for the Machine
M1. start:end_process::reset::start
M2. start:end_process:block:unblock::start
M3. start:end_process:reset::end_process
M4. start:end_process::start

5. Test-case implementation and evaluation

The buffer and the machine described in the example above, were developed in Java using the three-phase discrete-event simulation library JSim developed by M. Pidd [11]. JSim implements the three-phase approach [12] according to which the simulation proceeds through three phases: A-phase determines the next simulation time and forwards simulation clock to that time, B-phase executes the activities scheduled to be executed at the current simulation time and C-phase attempts to execute all the conditional activities.

In order to test the effectiveness of the test-cases generated in section 4, a number of errors were imputed in the code that implemented the Buffer and the Machine. Then, the test-cases generated in Section 4 were implemented in JUnit. The implementation of the equivalence-partitioning test-cases was straight-forward. For example, test case #4 for the buffer was implemented as follows (Figure 3): Initially a buffer (b) with capacity 2 and three parts (p1, p2, p3) are constructed. Then all the part are attempted to be added to the buffer and it is checked that the buffer contains exactly two part (i.e. parts p1 and p2).

```
public void testAddPartToNonEmptyBuffer2 () {
   b = new Buffer (2);
   p1 = new Part ("Part", 1);
   p2 = new Part ("Part", 2);
   p3 = new Part ("Part", 2);
   b.addPart (p1);
   b.addPart (p2);
   b.addPart (p3);
   assertTrue (b.size( ) == 2);
   assertTrue (b.contains(p1));
   assertTrue (b.contains(p2));
   assertTrue (!b.contains(p3));
 }
```

Figure 3: Implementation of the equivalence partitioning test case #4 for the Buffer in JUnit.

The implementation of the test-cases generated by X-Machine required a number of extensions to be introduced in the simulation library. More explicitly, a) a mechanism that allows the user to determine the first simulation phase and b) a facility to re-initiate the simulation system, so that simulation runs for the successive test-case could be executed in JUnit were developed. Furthermore, some X-Machine test-cases require the development of a function (stub) in order to achieve a desired result. For example, the test-case M2 requires the development of a function that will achieve the unblocking of the machine. The tester has both to develop this stub and to call it at the appropriate simulation time.

Table 2 shows the errors imputed and the test-case that identified them: thus, errors identified by the equivalence partitioning and by the X-Machine test-cases are shown in the 4th and 5th column of the table respectively.

No.	Method containing the error	Description of imputed error	Identified by EPT	Identified by XMT
ERRORS IMPUTED IN BUFFER				
1	Remove Part	Removing a part decreases buffer's capacity by one	--	✓
2	AddPart	Adding a part increases buffer's capacity by one	✓	✓
3	AddPart	A part already stored in the buffer is allowed to be added in it again	✓	--
4	AddPart	No part is added to the buffer	✓	✓
ERRORS IMPUTED IN MACHINE				
1	start	start could be called even if machine was not idle	✓	--
2	start	The part to be processed was not removed from the machine's input buffer	✓	✓
3	start	The end of processing was not scheduled	--	✓
4	start	The machine's state was not changed to processing	✓	✓
5	end	Machine status is not changed to idle at successful completion of the processing	✓	✓
6	end	Machine status is not changed to blocked when blockage occurs	✓	✓
7	Unblock	Part is not placed in the proper next buffer	✓	✓

Table 2: The errors imputed in the Buffer and Machine code and the test-cases that identified them.

6 Discussion

All the errors introduced in the code were identified by the test-cases generated the test-cases generated in Section 4. However, none of the two black-box testing approaches had the power to identify all the errors imputed in the code. Equivalence partitioning test-cases are capable of identifying errors whose effects may become obvious at the execution of a function. Buffer error #3 is an example of this type of error: when the addition of a new part is attempted, it is checked whether this part is already stored in the buffer. Errors related to the erroneous modification of memory values, on the other hand, are difficult to be identified at the time they are committed. Their presence may be identified by the erroneous output produced during the execution of another function which operates on the incorrectly set memory value. Buffer error #1 is an example of this type of error. Integration testing which tests the proper co-operation of the functions developed is more suitable to uncover this type of error. It is interesting to note that buffer error #2 is identified by both testing techniques (test case #4 of the equivalence partitioning approach and test case B5 of the X-Machine test-cases) . This is due to the fact that the implementation of the equivalence partitioning test-cases (#4) in JUnit closely resemble an integration test-case: the function add_part is called repetitively in order to set up the buffer's memory as required by the test-case.

7 Concluding remarks

Black-box testing techniques provide powerful test-cases based on system specifications, but unfortunately, no single black-box testing technique is powerful enough to uncover all coding errors. Attempts to combine alternative black-box testing techniques into one as suggested in [13] are interesting and worth-exploring especially if the appropriate test-specific software is developed.

The usage of test-specific software facilitates the adoption of the testing techniques and reduces time and effort: test-case generation, test-case development and testing itself may become quite time-consuming activities if not supported by the appropriate software.

The X-Machine approach is suitable for designing and testing stand-alone simulation entities. This approach is practical since it allows the design, implementation and testing of simulation entities with well-designed functionality.

Acknowledgment

The Project is co-funded by the European Social Fund and National Resources - (EPEAEK-II) ARHIMIDES.

References

[1] Rohrer, M. 1999. "Automod product suite tutorial", Proc. of the 1999 Winter Simulation Conference, Ed. P.A. Farrington, H.B. Nembhard, D.T. Sturrock, G.W. Evans., pp. 220-226.

[2] Price, R. N. and Harrell, C.R.. 1999. "Simulation modelling and optimisation using Promodel", Proc. of the 1999 Winter Simulation Conference, Ed. P.A. Farrington, H.B. Nembhard, D.T. Sturrock, G.W. Evans. pp. 208-214

[3] Sadowski, D. and Bapat, V. 1999. "The Arena product family: enterprise modelling solutions", Proc. of the 1999 Winter Simulation Conference, Ed. P.A. Farrington, H.B. Nembhard, D.T. Sturrock, G.W. Evans. pp. 159-166.

[4] O'Reilly. JJ and Lilegdon, W.R. 1999. "Introduction to FACTOR/AIM", Proc. of the 1999 Winter Simulation Conference, Ed. P.A. Farrington, H.B. Nembhard, D.T. Sturrock, G.W. Evans. pp. 201-207.

[5] Miller, J.A., Y. Ge and J. Tao. 1998. Component-Based Simulation Environments: JSIM as a Case Study Using Java Beans, In: Proceedings of the 1998 Winter Simulation Conference, ed. D. Medeiros, E. Watson, J. Carson, and M. Manivannan, 373-381, Washington, DC, 13-16 December.

[6] R.A. Kilgore, Object-oriented simulation with SML and SILK in .NET and Java, In: Proceedings of the 2003 Winter Simulation Conference, ed. S. Chick, P.J. Sanchez, D. Ferrin and D. J. Morrice, p. 218 – 224.

[7] Howell, F. and R. McNab. 1998. Simjava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling, In: Proceedings of the First International Conference on Web-Based Modeling and Simulation, San Diego, CA, January.

[8] G. J. Myers, The art of software testing, Wiley, 1979.

[9] Ipate F. and Holcombe M., "Specification and testing using generalised machines: a presentation and a case study", Software Testing, Verification and Reliability, Vol.8, 1998, pp. 61-81.

[10] Chow T.S., "Testing Software DesignModeled by Finite-State Machines," IEEE Transactions on Software Engineering, Vol.SE-4, No.3, 1978, pp.178-187.

[11] M. Pidd, Using Java to Develop Discrete Event Simulation", J. Opl. Res. Soc.

[12] Tocher K., The Art of Simulation, Van Nostrand Company, Princeton NJ, 1963.

[13] F. Ipate, 2004, Complete Deterministic Stream X–Machine Testing, Formal Aspects of Computing, 16, p. 374 – 386.