Functional Testing Techniques for Discrete-Event Simulation

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 testing is an important phase of a simulation study which aims to uncover errors in the simulation program. This paper presents the adoption of two complementary functional techniques for the testing of simulation programs that are built on top of discrete-event simulation-specific libraries. The power of each technique in uncovering imputed errors in the simulation program is demonstrated while it is shown that the adoption of the techniques requires the extension of the simulation library with new functionality.
Key-Words: - Simulation, Functional Testing, Equivalence Partitioning, X-Machines

1 Introduction

Computer simulation is a popular approach for the study of complex systems which requires the development a simulation program that mimics the system under study. The simulation program is based on a conceptual model that describes the system to be simulated at the required level of detail. Domain-specific simulators (e.g. [1], [2], [3], [4]) are user friendly environments that greatly reduce the effort required for the development of the conceptual model by representing the simulated system as data by automating the development of the corresponding simulation program. It is therefore clear that if a simulator is used for the simulation of a system, simulation program testing is not necessary since it may legitimately be assumed that the program generation capability of the simulator has been exhaustively tested by the simulator developer. Despite the productivity achieved by the domain-specific simulators, simulation experts quite often have to develop simulation code since existing simulators do not always provide the required functionality. In these cases, the simulation program is developed either in a simulation language (e.g. [5], [6], [7]) or in a general-purpose language using simulation-specific libraries (e.g. [8], [9], [10]) and is tested in order to establish that the conceptual model has been properly transformed to program. Simulation languages and simulation-specific libraries 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 some extend this is illustrated by the fact that in simulation literature testing is discussed in relation to validation and verification. Thus, nearly all methodological simulation papers present validation verification and testing (VV & T) as necessary activities that need to take place throughout the simulation development life cycle and provide a more or less widely acceptable categorization of VV & T techniques. It is noticeable that the VV & T techniques presented are usually presented as a whole, without distinguish between validation techniques, verification techniques and testing techniques. Furthermore, the majority of the simulation papers that deal VV & T 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, important issues related to testing simulation programs such as: the application of existing testing approaches to simulation or the required modification/extension of existing simulation libraries in order to support the various testing techniques for simulation are not discussed in the simulation literature.

This paper shows how two complementary testing techniques (each testing the program at a different level of abstraction) may be simultaneously applied for testing simulation programs developed in a general programming language in combination with a discrete-event simulation-specific library. A simple manufacturing facility is used as an example to demonstrate the application of the testing approaches in simulation. Sections 2 and 3 describe the manufacturing facility and its specification respectively. Section 4 presents the two testing

techniques. Section 5 discusses the implementation of the testing techniques as well as the way they are adopted for use in simulation and demonstrates the power of the techniques in identifying errors imputed in a simulation program.

Testing is the process of executing a program with the intent of finding errors [11]. Testing is generally carried out in two steps. During the first step, data sets that will be used for the testing are generated and the expected program output for each data test is determined. The generated data sets are called test-cases. The generation of the test-cases is crucial: effective testing requires careful selection of appropriate data sets so that the functionality of the program is satisfactorily tested. During the second testing step, the program under test is fed with the data sets and the produced output is compared with the expected one: any difference between the simulation-generated output and the expected output identifies an error in the simulation program. The second step may also require the addition of extra code in the program, so that appropriate description of the behaviour of the program under test is collected.

Two approaches may be used for the generation of the test-cases: according to functional (or black-box) testing techniques [11] base the generation of test-cased on the specification (i.e. the conceptual model) 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 functional 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. A number of modelling techniques have been proposed for the development of conceptual models. Some of them, such as the Activity Cycle Diagram and Petri-Nets, are diagrammatic. An evaluation of them may be found in [12]. Diagrammatic models are quite popular since they are intuitive and quite easy to use. Despite these advantages, however, diagrammatic models provide no support for the testing stage. In this work the X-machine formalism has been adopted for the development of the conceptual models.

X-machines is a specification formalism introduced by Eilenberg [13], which is capable to model both the data and the control of a program. Thus, X-machines employ a diagrammatic approach to model the control by extending the expressive power of the Finite State Automata (FSA). Transitions between states are no more performed through simple input symbols but through the application of functions. These functions are written in a formal notation and model the processing of the data. Data, on the other hand, is held in memory, which is attached to the X-machine. Functions receive input, read the memory values, and produce output while modifying the memory values. The conceptual model (developed

as an X-Machine) of the system described above is presented next.

3.1 X-Machine Buffer specification

The Finite State Automaton that corresponds to the buffer specification is shown in Figure 1:

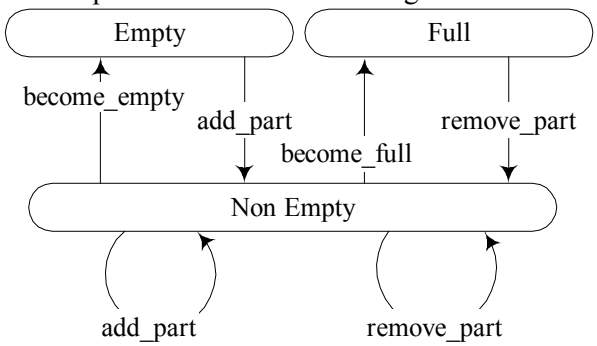

Figure 1: The associated finite state automaton for the Buffer. Initial state: Empty.

The functions add_part and become_full are responsible for adding a new part into the buffer while remove_part and become_empty remove a part stored in the buffer.

The buffer memory is: BM = (PARTS × Capacity) where PARTS represents the set of 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.

The X-Machine functions that are shown in the Finite State Automaton will be specified in detail using following the notation:

f (inp, mem) = (outp, new_mem) guard

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

The Buffer X-Machine functions using the above notation are defined as follows:

add_part (p, (Parts, Capacity)) =
(part_added, (Parts ∪ p, Capacity)
if p ∉ Parts ∧ card (Parts) < Capacity – 1

become_full (p, (Parts, Capacity)) =
(part_added, (Parts ∪ p, Capacity)
if p ∉ Parts ∧ card (Parts) = Capacity – 1

remove_part (p, (Parts, Capacity)) =
(part_removed, ((part – p), Capacity))
if p ∈ Parts ∧ card (Parts) > 1

become_empty (p, (Parts, Capacity)) =
(part_removed, ((part – p), Capacity))
if p ∈ Parts ∧ card (Parts) = 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.

3.2. Machine specification

The Finite State Automaton that corresponds to the machine specification is shown in Figure 2.

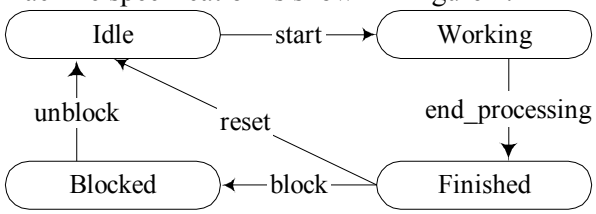

Figure 2: The associated finite state automaton for the Machine. Initial state: Idle.

The machine memory is:
MM = (Time× IN × PROC × OUT × BTime × 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: which represent the blockage of the machine due to full output buffer and its corresponding availability when the output buffer may accept another part.

The definition of these functions is given next:

start (check_start, (now, in, nil, out, nil, dur)) =
(proc_starts, (now, (in-p), p, out, now+dur, dur))
if (in ≠ empty ∧ Proc = nil ∧ BTime = nill)

end_processing (end, (now, in, p, out, t, dur)) =
(end_processing, (now, in, p, out, nil, dur))
if (now = t)

reset (out_buf_not_full, (now, in, p, out, nil, dur)) =
(proc_ended, (now, in, nil, out + p, nil, dur))

block (out_buf_full, (now, in, p, out, nil, dur)) =
(machine_blocked, (now, in, p, out, nil, dur))

unblock (out_buf_not_full, (now, in, p, out, nil, dur)) =
(machine_unblocked, (now, in, nill, out+p, nil, dur))

The complete conceptual model that has been developed using the X-Machine specification serves two aims: firstly, it allows the simulation developer to understand in detail the logic of the system to be simulated and secondly, it supports the testing of the simulation program as described next.

4 Test-case generation

In this section two complementary functional testing techniques may be used for the testing of simulation entities are presented. The first functional 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 valid and invalid equivalence classes of test data are generated: valid equivalence classes represent valid values to variables while invalid equivalence classes represent erroneous variable values. The equivalence classes are identified based on the specification of the functions.

add_part (p, (Parts, Capacity)) =
(part_added, (Parts ∪ p, Capacity)
if p ∉ Parts ∧ card (Parts) < Capacity – 1

According to the guard of the function add_part a part p that is not currently hold in the buffer (p ∉ parts) may be added in the buffer when the buffer has more than one empty space (card (parts) < capacity – 1). A valid test-case that represents this situation is to add part p2 to a buffer with capacity 5 that holds the part p1. This valid test-case may be described by the buffer memory together with the part to be added to the buffer using the notation: MB = (<p1>, 5), p = p1. Invalid equivalence classes for the function add_part need also be derived: one invalid equivalence class represents the case where a part already stored in the buffer is attempted to be added again in the buffer. This invalid equivalence class is represented by the data set: MB = (<p1>, 5), p = p1; another invalid equivalence class is when the buffer has not more than one empty space to hold the part and is represented by the data set: MB = (<p1, p2>, 2), p = p3. The valid and invalid equivalence classes for the functions of the Buffer, together with their corresponding test-cases are shown in the Table 1.

Function: add_part

valid equivalence classes	
p ∉ parts ∧ card (parts) < capacity – 1	MB = (<p1>, 5), p = p2
invalid equivalence classes	
p ∈ parts ∧ card (parts) < capacity – 1	MB = (<p1>, 5), p = p1
p ∉ parts ∧ card (parts) > capacity –1	MB = (<p1, p2>, 2), p = p3
p ∉ parts ∧ card (parts) = capacity –1	MB = (<p1>, 2), p = p2

Function: become_full

valid equivalence classes	
p ∉ parts ∧ card (parts) = capacity – 1	MB = (<p1>, 2), p = p2
invalid equivalence classes	
p ∈ parts ∧ card (parts) = capacity – 1	MB = (<p1>, 2), p = p1
p ∉ parts ∧ card (parts) > capacity –1	MB = (<p1, p2>, 2), p = p3
p ∉ parts ∧ card (parts) < capacity –1	MB = (< >, 2), p = p3

Function: remove_part

valid equivalence classes	
p ∈ Parts ∧ card (Parts) > 1	MB = (<p1, p2>, 2), p = p1
invalid equivalence classes	
p ∉ parts ∧ card (parts) > 1	MB = (<p1, p2>, 2), p = p3
p ∈ parts ∧ card (parts) = 1	MB = (<p1>, 2), p = p1
p ∈ parts ∧ card (parts) < 1	MB = (<p1>, 3), p = p1

Function: become_empty

valid equivalence classes	
p ∈ Parts ∧ card (Parts) = 1	MB = (<p1>, 2), p = p1
invalid equivalence classes	
p ∉ parts ∧ card (parts) = 1	MB = (<p1>, 2), p = p3
p ∉ parts ∧ card (parts) = 1	MB = (<p1>, 2), p = p3
p ∈ parts ∧ card (parts) > 1	MB = (<p1, p2>, 2), p = p1

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 [14] and is presented in detail in [15]. 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. For example, for the buffer, the cover set and the characterisation set are:

S = {e, add_part, add_part · become_full}
W = {become_empty, ignore_add}

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 [16]. JSim implements the three-phase approach [17] according to which the simulation evolves through the execution of two types of activities: bound to time (B-activities) which may be scheduled in advance and system-state dependent activities (C-activities) which are executed when specific system conditions are met. Three-phase executives cycle through three phases: A-phase determines the next simulation time and forwards the simulation clock to that time, B-phase executes the B-activities scheduled to be executed at the current simulation time and C-phase attempts to execute all the conditional activities.

Initially, the code that implements the Buffer class was developed in Java (Figure 3)

```
public class Buffer extends Queue  { //Queue is a Vector
   String name;
   int capacity;

Buffer (int c) {
            super( );
            name = "unnamed buffer";
            capacity = c;
            }

public boolean isFull ( ) {
            if (capacity == size()) return true;
            else return false;
            }

public boolean remove (Part p) {
            return remove (p);
            }

public boolean add (Part p) {
            if (size( ) <= capacity - 1 && !contains (p)) {
                        addElement(p);
                        return true;
            }
            else return false;
}
```

Figure 3: an extract of the code for Buffer

The class Queue is provided in JSim library and is an extension of the Java class Vector. It should be also noted that buffer X-machine functions add_part and become_full are combined into a single method (named add) while the functions remove_part and become_empty are combined into a single Java method named remove.

Next, the class Machine was implemented in Java using the JSim facilities. The logic of the B- and C-activities of the Machine.java is shown in Figure 4:

C-activity: **startProcessing**
IF the machine is idle AND its input buffer contains parts THEN
    get a part from the machine's input buffer and place it in the processing buffer
    set machine status to processing
    schedule the end of processing for this machine

B-activity: **endProcessing**
IF the machine is processing THEN
    set machine status to finished
    find the next buffer for the processed part
    IF the next buffer is not full THEN
        move the part from the processing buffer to the next buffer
        set machine status to idle
    ELSE
        set machine status to blocked

C-activity: **unblock**
IF the machine is blocked THEN
    find the next buffer for the processed part
    IF the next buffer is not full THEN
        move the part from the processing buffer to the next buffer
        set machine status to idle

Figure 4: the logic of the B- and C-activities of the Machine

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. JUnit is an open source Java testing framework used to write and run repeatable tests. The implementation of the equivalence-partitioning test-cases in JUnit was straight-forward. For example, test case #4 for the buffer was implemented as follows (see also Figure 5): 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 parts (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 5: 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. These extensions where: a) a mechanism that allows the user to determine the initial simulation phase and b) a facility to re-initiate the simulation system, so that simulation runs corresponding to successive test-cases 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

The combined application of the presented functional testing techniques, as suggested for example in [18], uncovered all the errors imputed in the simulation code. Each functional technique achieved to uncover different types of errors. Thus, the equivalence partitioning test-cases was capable of identifying errors that violated function-level logic. For example, the addition of a part in a buffer that already stores that part (the third error imputed in the Buffer) is detected by the second invalid test-cases of the functions add_part and become_full of the buffer. Errors of this type usually have immediate effects that become obvious during the execution of a single function. X-Machine test-cases, on the other hand, are more suited in uncovering errors that violate system-level logic. The erroneous modification of a memory value, for example, without any other immediately obvious symptoms, is such a type of error that may go undetected for long execution time after its commission. The presence of those errors is identified during the execution of another function which utilises the incorrectly set memory value. The decrease of the buffer's capacity by one during the removal of a part from the buffer (the first error imputed in the Buffer) is an error that may be identified by the X-Machine test-case: add_part remove_part add_part become_empty when applied on a buffer with capacity two, since at the second call of add_part, the buffer will be found full given that its capacity will have been reduced to one during the execution of become_empty.

The importance of employing good programming practices should also be stressed. For example, the first error imputed in the Buffer, could be identified by the java compiler if the buffer capacity had been declared as final. In addition, the extent to which the system state is checked through assertions is also a point of concern. For example, the first error imputed in the Buffer might have been identified had the assertTrue (b.getCapacity( ) == 2) been included in the assertions.

It is interesting to note that none of the two functional techniques has the power to identify all the imputed errors, thus a combination of techniques as suggested for example in [18] seems to be the most efficient approach.

The employment of testing frameworks greatly facilitates the implementation of the testing techniques and reduces programmers' 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.

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] P.J. Kiviat, R. Villanueva, H.M. Markowitz, The SIMSCRIPT II Programming Language, Prentice-Hall, New Jersey, 1968

[6] A. T. Clementson, ECSL – Extendd Control and Siumulation Users Manual, CLE.COM Ltd Birmingham, England, 1985.

[7] G. M. Birtwhistle, O.J. Dahl, B. Myhrhaug, K Nygaard, SIMULA Begin, 2nd edition, Von Nostrand Reinhold, NY, 1979.

[8] 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.

[9] 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.

[10] 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.

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

[12] Page E., *Simulation modelling methodology: principles and etiology of decision support*, PhD Thesis, Dept. of Computer Science, Virginia Polytechnic Institute, 1994.

[13] Eilenberg S., *Automata Machines and Languages*, Vol. A, Academic Press, 1974.

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

[15] 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.

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

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

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