Paweł Skruch*

Application of Model-Based Approach for Testing Dynamic Systems

1. Introduction

In today's world, systems that contain software are everywhere. They may be observed in common devices employed in everyday living (e.g., coffee machines, washing machines, cell phones) as well as in sophisticated engineering systems (e.g., cars, planes, spacecrafts). As the complexity of control systems grows, testing becomes more and time consuming. Poorly tested systems may cost producers billions of dollars annually especially when defects are found by end users in production environments. Barry Boehm's research analysis [5] indicates that the cost of removing a software defect grows exponentially for each stage of the development life cycle in which it remains undiscovered. Boris Beizer [2] estimates that 30 up to 90 percentage of the effort in put into testing. Another research project conducted by the United States Department of Commerce, National Institute of Standards and Technology [21] estimated that software defects cost the U.S. economy $60 billion per year.

There are several facts that show clearly possible consequences of poorly tested systems. On February 25, 1991, an Iraqi Scud hit the barracks in Dhahran in Saudi Arabia, killing 28 soldiers from the US Army. This accident was caused by software error in the system's clock [20]. The Patriot missile battery has been in operation for 100 hours, by which time the system's internal clock had drifted by one third of a second. For a target moving as fast as Scud, this was equivalent to a position error of 600 meters. Another example is connected with Therac-25 radiation therapy machine that was produced by Atomic Energy of Canada Limited and CGR of France. The machine was involved with at least six known accidents between 1985 and 1987, in which patients were given massive overdoses of radiation, which were in some cases on the order of hundreds of grays [13]. At least five patients died of the overdoses. These accidents were caused by errors in software control application. One of the most infamous computer bugs in history was found during flight 501 that took place on June 4, 1996. This was the first, and unsuccessful, test flight of the European Ariane 5 expendable lunch system. Due to an error in the software design (inadequate protection from integer

* AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Automatics, al. Mickiewicza 30, 30–059 Krakow, Poland. E-mail: pawel.skruch@agh.edu.pl

overflow), the rocket veered off its flight path 37 seconds after launch and was destroyed by its automated self-destruct system [12]. As it was an unmanned flight, there were no victims, but the breakup caused the loss of four Cluster mission spacecraft, resulting in a loss of more than $370 million.

The paper is organized as follows. In the Section 2, a model-based approach for system development is briefly described. Then, in the Section 3, the concept of testing with model usage as an oracle is presented. Section 4 describes the mathematical model for a continuous-time dynamic system. Section 5 is devoted to mathematical and implementation challenges associated with testing of the dynamic systems. Basic concepts and ideas are investigated and concluded. An application example is given in Section 6. Conclusions are in Section 7.

2. System development using model-based design

The extensive use of electronics and software within the last decades has revolutionized the implementation and scope of many of the features and functions in modern control systems. This electronic revolution has significantly improved the performance, quality and reliability of the systems and, in parallel, has resulted in a staggering complexity. Model-based design (MBD) [9,14] is becoming the preferred methodology for the development of software systems because of its capability to address the corresponding system complexity and productivity challenges.

Models are used in engineering disciplines to specify the system's behavior in a clear and unambiguous form. The selection of the most appropriate model is crucial since this selection determines the understanding of the functionality of the system and influences further steps in the implementation of the system. Finite state machines (FSMs) and Petri nets [7] are used to capture control oriented systems and provide useful mathematical ways of verifying their correctness. Data flow graphs and flowcharts [7] are used in the data dominated applications domains. Boolean algebra [24] is usefulness in modeling automatic control devices. Dynamic behavior is modeled using difference or differential equations [15]. The current state of the art is that there are efficient testing algorithms and methods for the systems modeled using Boolean algebra or graphs (state charts, finite state machines) [3,4,16] but testing of the dynamic systems modeled by differential equations is relatively poorly supported by tools and methods [19,26].

Figure 1 illustrates a possible development scenario that utilizes the modelling approach. The traditional means of specifying the system's behavior is a mixture of textual description, mathematical and logical relationships. This form is often called requirement specification. A model is built based on the requirements and it can reside at different levels of abstraction. It can consider the main functional aspects of the system's behavior as well as address various target system implementation constraints such as real-time operating system, scheduling details, signal representation, fixed-point versus floating-point number representation, etc. The model is used in computer simulations in an early stage of development to validate the system concept, calibrate parameters and optimize the system performance.

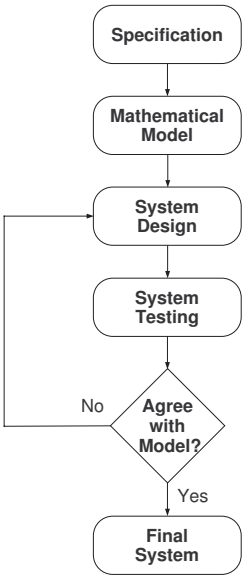

Fig. 1. Simplified flowchart for model-based design methodology

The purpose of system design is to create a technical solution that satisfies the requirements for the system. The specification is transformed into a physical architecture, software and hardware components are designed and prototypes. The code can be either manually created based on the model or auto-generation can be used to get code automatically generated from the model. In the next step, the code is downloaded to the hardware and the entire system is being tested during system testing phase. Testing process allows verifying that the software behavior in the final hardware is identical to that observed during computer simulations. If the tests are failed then the system needs to be redesigned that is either software or hardware components need to be corrected.

3. Concept of model-based testing

Model-based testing (MBT) [19,23,26] refers to the use of models to generate tests and develop a test oracle mechanism [1] for software components or entire systems. The model in the MBT concept is considered as a theoretical construction that contains a detailed and true description of the physical system using mathematical relationships and equations. Tests either are developed manually or are automatically generated from the model, which represents expected behavior of the system under tests (SUT). The model can be also used as an oracle (Fig. 2), that is as a mechanism to judge if the results of an executed test case qualify as passed or failed.

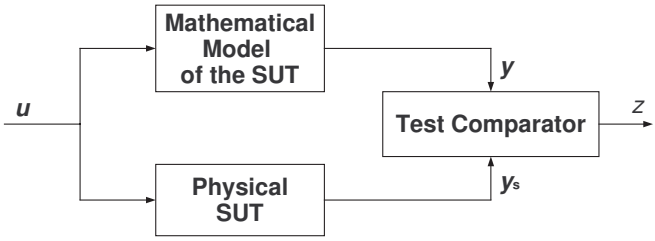

Fig. 2. Testing with a model as an oracle

The execution of a test case consists of exciting the system using actuators to simulate its working conditions, and measuring the system's response in terms of electrical signals, motion, force, strain, etc. The signals are physical in case of the SUT and virtual in case of the model. The approach stipulates that the same inputs ($\boldsymbol{u}$) are applied to both the SUT and to the model. Next, the responses from the SUT ($\boldsymbol{y}_s$) and from the model ($\boldsymbol{y}$) are compared by a test comparator to determine whether a test case has passed or failed.

4. Mathematical model of the system under test

Dynamic systems have output responses to inputs that may continue after the inputs are held constant. The outputs of the system depend not only on the current state of the inputs, but also on the history of those inputs (which is preserved and represented by different internal states of the system). Dynamic systems can respond to input signals or initial conditions (Fig. 3).

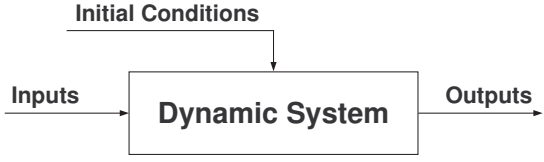

Fig. 3. Elements of a dynamic system

The dynamic behavior of a finite-dimensional, time-invariant system with continuous-time parameter can be specified by the state space model (Fig. 4):

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \quad \boldsymbol{x}(0) = \boldsymbol{x_0}, \tag{1}$$

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), t), \tag{2}$$

where $\boldsymbol{x}(t) \subset X \in \mathbb{R}^n$ refers to the system state, $\boldsymbol{u}(t) \subset U \in \mathbb{R}^r$ refers to the system input, $\boldsymbol{y}(t) \subset Y \in \mathbb{R}^m$ refers to the system output, the independent variable $t > 0$ is time, $\boldsymbol{x_0} \in \mathbb{R}^n$ is the given initial condition, $\boldsymbol{f} : \mathbb{R}^n \times \mathbb{R}^r \times \mathbb{R} \to \mathbb{R}^n$ denotes a mathematical relationship describing the system behavior, $\boldsymbol{g} : \mathbb{R}^n \times \mathbb{R}^r \times \mathbb{R} \to \mathbb{R}^m$ determines the output, X is the state space, U is the input space, Y is the output space, $\mathbb{R}^n$, $\mathbb{R}^r$, $\mathbb{R}^m$ are real vector spaces of column vectors, n, r, m are positive integers.

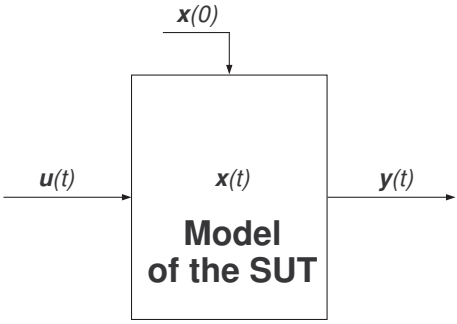

Fig. 4. State space modelling concept of the SUT

5. Challenges in testing of dynamic systems

In this section, mathematical and implementation challenges associated with testing of the dynamic systems are briefly described. These challenges are related to notation of tests, calculation of test coverage, implementation of a test comparator and automatic generation of test cases. Some author's ideas and solutions are presented.

5.1. Notation of tests

A test case can be considered as a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [22]. Using this definition with the state space modeling concept of the SUT (1), (2), a single test case $\mathrm{T}_{\mathrm{case}}^{(j)}$ can be defined as:

$$\mathrm{T}_{\mathrm{case}}^{(j)}: \quad \left(T^{(j)}, \boldsymbol{x_0}^{(j)}, \boldsymbol{u}^{(j)}, \boldsymbol{x}^{(j)}, \boldsymbol{y}^{(j)}\right), \tag{3}$$

where $\boldsymbol{u}^{(j)} : [0, T^{(j)}] \to \mathbb{R}^m$ is an input function applied to the SUT, $\boldsymbol{x}^{(j)} : [0, T^{(j)}] \to \mathbb{R}^n$ is an expected state function and $\boldsymbol{y}^{(j)} : [0, T^{(j)}] \to \mathbb{R}^m$ is an expected output function when the system starts from an initial condition $\boldsymbol{x_0}^{(j)}$, $j = 1, 2, \ldots, N$ is a label to indicate different test cases. In the notation (3) expected outcomes for the test case $\mathrm{T}_{\mathrm{case}}^{(j)}$ are the state function

$\boldsymbol{x}^{(j)}$ and the output function $\boldsymbol{y}^{(j)}$. The design of test cases can utilize at least one of these outcomes depending on what is available in test environment. A collection of one or more test cases forms a test suite $\mathrm{T}_{\mathrm{suite}} = \left\{ \mathrm{T}_{\mathrm{case}}^{(1)}, \mathrm{T}_{\mathrm{case}}^{(2)}, \ldots, \mathrm{T}_{\mathrm{case}}^{(N)} \right\}$.

5.2. Implementation of a test comparator

The test comparator can be considered as a tool that implements a mechanism for determining whether a test has passed or failed. In the MBT concept (Fig. 2), this tool compares the actual output produced by the SUT with the expected output produced by the model. Possible realizations of the comparison function z for a given test case $\mathrm{T}_{\mathrm{case}}^{(j)}$ are presented below:

$$z(\mathrm{T}_{\mathrm{case}}^{(j)}) = \begin{cases} 0 & \text{if } \forall_{t \in [0, T^{(j)}]} \left\| \boldsymbol{y}^{(j)}(t) - \boldsymbol{y}_{\mathrm{s}}^{(j)}(t) \right\| < \varepsilon, \\ 1 & \text{otherwise,} \end{cases} \tag{4}$$

$$z(\mathrm{T}_{\mathrm{case}}^{(j)}) = \begin{cases} 0 & \text{if } \frac{1}{T^{(j)}} \int_0^{T^{(j)}} \left\| \boldsymbol{y}^{(j)}(t) - \boldsymbol{y}_{\mathrm{s}}^{(j)}(t) \right\|^2 \mathrm{d}t < \varepsilon, \\ 1 & \text{otherwise,} \end{cases} \tag{5}$$

$$z(\mathrm{T}_{\mathrm{case}}^{(j)}) = \begin{cases} 0 & \text{if } \left\| \boldsymbol{y}^{(j)}(T^{(j)}) - \boldsymbol{y}_{\mathrm{s}}^{(j)}(T^{(j)}) \right\| < \varepsilon, \\ 1 & \text{otherwise.} \end{cases} \tag{6}$$

The comparison function z can be defined in many ways assuming that $z = 0$ if the actual output is within a predefined tolerance range ε relative to the expected output (then the test is qualified as passed), and $z = 1$ in other cases (then the test is qualified as failed). The tolerance range ε should be chosen based on the system requirements and according to the characteristics of the tested system.

5.3. Calculation of test coverage

Test coverage is a way to characterize the relation between the number and the type of tests to execute and the portion of the system's behavior effectively tested. The purpose of a test coverage analysis is then to indicate when sufficient testing has been performed and where additional testing is needed. The most obvious quantification of the system's behavior exercised by a test suite is computed by dividing the number of the system states explored by the test suite by the cardinality of the entire output space. However, this method has limited usefulness since the output space for dynamic systems has usually infinite number of states. In such situation, one of the possible ways out is to transform the output space Y into another one $Y_{\boldsymbol{h}}$ that contains countable number of elements (see also [17]). The test coverage $C_{\boldsymbol{h}}$ of

the test suite $\mathrm{T}_{\mathrm{suite}} = \left\{\mathrm{T}_{\mathrm{case}}^{(1)}, \mathrm{T}_{\mathrm{case}}^{(2)}, \ldots, \mathrm{T}_{\mathrm{case}}^{(N)}\right\}$ can be then defined as follows [18]:

$$C_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}}) = \frac{\left|\bigcup_{j=1}^{N} V_{\boldsymbol{h}}\left(\mathrm{T}_{\mathrm{case}}^{(j)}\right)\right|}{|Y_{\boldsymbol{h}}|}, \tag{7}$$

where

$$Y_{\boldsymbol{h}} = \left\{\boldsymbol{i} \in \mathbb{Z}^m : \exists_{\boldsymbol{y} \in Y} : \boldsymbol{y} \in G_{\boldsymbol{h}}(\boldsymbol{i})\right\}, \tag{8}$$

is the transformed output space, $\boldsymbol{h} = [h_1\, h_1 \ldots h_m]^{\mathrm{T}}$, $h_k > 0$ for $k = 1, 2, \ldots, m$, $\mathbb{Z}$ stands for the set of integers,

$$G_{\boldsymbol{h}}(\boldsymbol{i}) = \left\{\boldsymbol{y} \in \mathbb{R}^m : \boldsymbol{y} = [y_1\, y_2 \ldots y_m]^{\mathrm{T}},\ \left\lfloor \frac{y_k}{h_k} \right\rfloor = i_k,\ k = 1, 2, \ldots, m\right\}, \tag{9}$$

denotes a hyperelement in the space $\mathbb{R}^m$, $\left\lfloor \frac{y_k}{h_k} \right\rfloor$ is the largest integer not greater than $\frac{y_k}{h_k}$,

$$V_{\boldsymbol{h}}\left(\mathrm{T}_{\mathrm{case}}^{(j)}\right) = \left\{\boldsymbol{i} \in Y_{\boldsymbol{h}} : \exists_{t \in [0, T^{(j)}]} : \boldsymbol{y}^{(j)}(t) \in G_{\boldsymbol{h}}(\boldsymbol{i})\right\} \tag{10}$$

is the number of states of the transformed output space covered by the test case $\mathrm{T}_{\mathrm{case}}^{(j)}$.

The proposed test coverage measure is defined using a partition of the system output space. The partition forms a rectangular grid and, roughly speaking, the test coverage is defined by the number of the grid boxes visited by the output during a test.

5.4. Selection of tests

In spite of continuing research on test approaches for continuous and mixed discrete-continuous systems (for example [6,8,10,25,26]), there is still a lack for patterns, processes, methodologies, and tools that effectively support automatic generation and selection of the correct test cases for such systems. A promising approach uses time- and frequency-domain techniques borrowed from system identification methodology. The key to obtaining a worthwhile result in system identification is choosing recognizable test signals. One of the popular practices is to use a series of single variable step tests. Every input to the system is stepped separately and some step responses are expected on the system output. Another research [6] shows that the Rapidly-exploring Random Tree RRT algorithm [11] can successfully solve testing problems for hybrid systems. The basis for the method is the incremental construction of search trees that attempt to rapidly and uniformly explore the state space. The proposition provided in [10] allows generating tests from hybrid systems' models with the help of robust test notion. The core idea of this proposal is to compute the robust neighborhood around a

given initial state. Such neighborhood consists of initial states that have the same qualitative behaviors. Using this notion, the trajectories of the system can be appropriately constructed.

6. Illustrative example

Consider a simple system (Fig. 5) that is responsible for monitoring fuel consumption and optimizing the refueling of a vehicle. The fuel level in the tank is not easily converted into an accurate indication of the available fuel volume. This relationship is strongly affected by the irregular shape of the fuel tank, dynamic conditions of the vehicle (accelerations, braking, etc.) and oscillations of the indication provided by the sensors. The main elements of the sys-

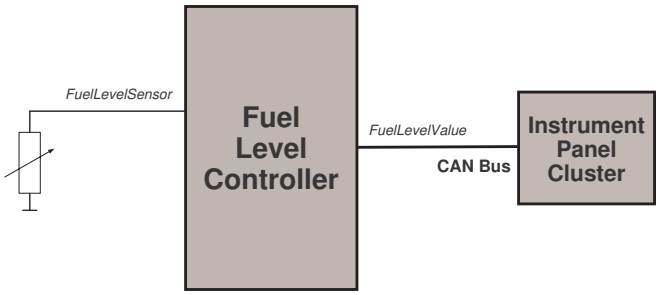

Fig. 5. Functional diagram of a fuel level control system

tem are: a fuel level sensor designed to be mounted on the fuel tank, a Fuel Level Controller (FLC) and an Instrument Panel Cluster (IPC). The fuel level sensor is a potentiometer and it provides a continuous resistance output proportional to the fuel level in the tank. If the tank is full, the sensor shall provide to the control module the resistance of $100\,\Omega$. If the tank is empty, the sensor shall provide the resistance of $0\,\Omega$. The control module shall filter the input signal (*FuelLevelSensor*) and then transmit via CAN bus the filtered value (*FuelLevelValue*) to the IPC, which is responsible for fuel level indication to the driver. The filtered value can be the result of the following calculation:

$$\dot{x}(t) = -\frac{1}{T}x(t) + \frac{1}{T}u(t), \quad x(0) = x_0, \tag{11}$$

$$y(t) = x(t), \tag{12}$$

where $u(t)$ represents the resistance provided by the sensor at time $t > 0$, $y(t)$ denotes the filtered value, $x(t)$ is the state variable in the system, x_0 is the initial condition.

Many software systems (e.g., real-time embedded systems) have multiple resource constraints such as energy, memory, and implementation constraints. In this example, the fuel level sensor provides the resistance from the specified range (i.e., from 0 to $100\,\Omega$). The indication of available fuel shall vary from 0 to 100 %. These constraints can be reflected in the

formulation that $u(t) \in [0,100]$, $x(t) \in [0,100]$, $y(t) \in [0,100]$ for all $t \geq 0$. Thus, the spaces X, Y and U can be considered as bounded, smooth, and compact manifolds in the space $\mathbb{R}$.

Consider a single test case

$$\mathrm{T}^{(1)}_{\mathrm{case}}: \quad \left(T^{(1)}, x_0^{(1)}, u^{(1)}, y^{(1)}\right), \tag{13}$$

where

$$T^{(1)} = 2000, \tag{14}$$

$$x_0^{(1)} = 0, \tag{15}$$

$$u^{(1)}(t) = 100\chi_{[0,\infty)}(t), \tag{16}$$

$$y^{(1)}(t) = 100\left(1 - \exp\left(-\frac{t}{T}\right)\right). \tag{17}$$

The step function (16) shall be applied both to the input of the SUT and its functional model (11), (12). The step response (17) is expected on the system output. Figure 6 shows a comparison of the responses obtained from the SUT and the model.

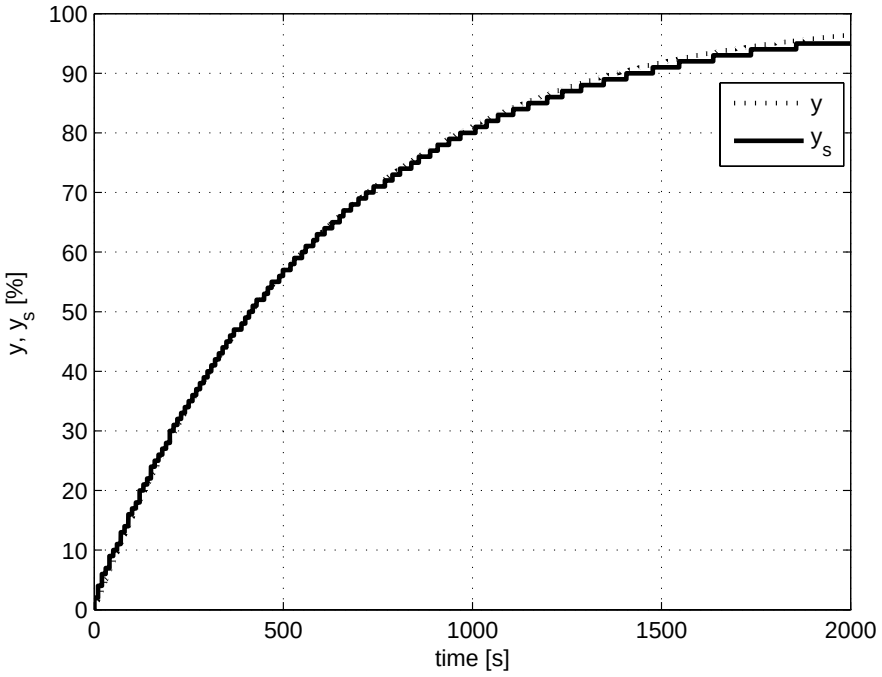

Fig. 6. Comparison of the outputs from the SUT and its model in reaction on the same input

Table 1 presents results of the comparison function (4) for the given test case $\mathrm{T}_{\mathrm{case}}^{(1)}$. Depending on the tolerance range ε the same test case can be qualified as passed ($\varepsilon = 3.0$) or failed ($\varepsilon = 2.0$ or $\varepsilon = 3.0$). Table 2 presents the similar results for the comparison functions (5) and (6). It should be noted that right choice of the comparison function as well as required tolerance range might have big influence on the entire development and test processes.

Table 1
Values of the comparison function (4) for the test case (13)

Tolerance ε	$z(\mathrm{T}_{\mathrm{case}}^{(1)})$ (Eq. (4))
2.0	1
2.5	1
3.0	0

Table 2
Values of the comparison functions (5) and (6) for the test case (13)

Tolerance ε	$z(\mathrm{T}_{\mathrm{case}}^{(1)})$ (Eq. (5))	Tolerance ε	$z(\mathrm{T}_{\mathrm{case}}^{(1)})$ (Eq. (6))
45	1	0.5	1
60	1	1.0	1
75	0	1.5	0

The output space $Y = [0,100] \subset \mathbb{R}$. Consider then the transformed output space Y_h defined by (8) for $h = 1$. The space Y_h contains a countable number of elements:

$$Y_h = \{0,1,2,\ldots,100\}, \tag{18}$$

and the hyperelement G_h is an interval:

$$G_h(i) = [i, i+1). \tag{19}$$

According to Fig. 6, the following elements of the space Y_h are covered by the test case $\mathrm{T}_{\mathrm{case}}^{(1)}$:

$$V_h\left(\mathrm{T}_{\mathrm{case}}^{(1)}\right) = \{0,1,2,\ldots,96\}. \tag{20}$$

The coverage for the single test case $\mathrm{T}_{\mathrm{case}}^{(1)}$ is equal:

$$C_h(\mathrm{T}_{\mathrm{case}}^{(1)}) = \frac{\left|V_h\left(\mathrm{T}_{\mathrm{case}}^{(1)}\right)\right|}{|Y_h|} = \frac{97}{101} \approx 0.96. \tag{21}$$

7. Conclusions

MBD can be considered today as state of the art in many areas of engineering. MBT is a related feature that supports effective and efficient testing. Dynamic system represents also a wide class of systems that are implemented in digital machines. In order to assure good quality of the final systems, testing is necessary. The need for test methodologies is mainly motivated by the fact that formal proofs are required to document that the system meets requirements. It should be noticed that testing continuous aspects of control systems creates a number of challenges and still is not sufficiently supported by commercial-off-the-shelf tools. In addition, the most popular testing methods such as equivalence partitioning or boundary value analysis are typically insufficient for such systems. These methods check the system for some discrete magnitude values and discrete moments of time that are seldom representative. Testing continuous aspects of control systems requires test cases that utilize time continuous input signals and time continuous output signals. In the paper, a model-based approach has been presented that can help the test engineer in testing of continuous-time dynamic systems.

Acknowledgements

This work was supported by the National Science Centre (Poland) — project No N N514 644440.

References

[1] Adrion W., Branstad M., Cherniabsky J. (1982). Validation, verification and testing of computer software. *Computing Surveys*, **14**(2), 159–192.

[2] Beizer B. (1990). *Software Testing Techniques. 2nd ed.*. Van Nostrand Reinhold, Boston.

[3] Beizer B. (1995). *Techniques for Functional Testing of Software and Systems*. John Willey & Sons, New York.

[4] Binder R. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Boston.

[5] Boehm B. (1981). *Software Engineering Economics*. Prentice Hall, Englewood Cliffs.

[6] Dang T., Nahhal T. (2009). Coverage-guided test generation for continuous and hybrid systems. *Formal Methods in System Design*, **34**(2), 183–213.

[7] Gajski D.D., Vahid F., Narayan S., Gong J. (1994). *Specification and Design of Embedded Systems*. Prentice Hall, Englewood Cliffs.

[8] Hahn G., Philipps J., Pretschner A., Stauner T. (2003). *Tests for mixed discrete-continuous systems*. Technical Report TUM-I0301, Institut für Informatik, Technische Universität München.

[9] Helmerich A., Koch N., Mandel L. (2005). *Study of worldwide trends and R&D programmes in embedded systems in view of maximising the impact of a technology platform in the area.* Final Report for the European Commission, November 18, Brussels, Belgium.

[10] Julius A., Fainekos G.E., Anand M., Lee I., Pappas G. (2007). Robust test generation and coverage for hybrid systems. [in:] *Proceedings of the 10th International Conference on Hybrid Systems: Computation and Control (HSCC)*, April 2007, Pisa, Italy, 329–342.

[11] LaValle S.M., Kuffner J.J. (2001). Rapidly-exploring random trees: Progress and prospects. [in:] Donald B.R., Lynch K.M., Rus D. (Eds.), *Algorithmic and Computational Robotics: New Directions*, A K Peters, Welleslay, USA, 293–308.

[12] Lions J.L. (1996). *ARIANE 5. Flight 501 failure.* Ariane 501 Inquiry Board Report, Paris.

[13] Leveson N.G., Turner C.S. (1993). An investigation of the Therac-25 accidents. *IEEE Computer*, **27**(7), 18–41.

[14] MathWorks^{TM} : Model-based design. http://www.mathworks.com/model-based-design (accessed on December 08, 2010).

[15] Mitkowski W. (1991). *Stabilization of Dynamic Systems*. WNT, Warszawa.

[16] Myers G. (1979). *The Art of Software Testing*. John Willey & Sons, New York.

[17] Skruch P. (2010). A coverage metric for the verification of discrete-time dynamic systems. [in:] *Proceedings of the XII International PhD Workshop OWD 2010*, 23–26.10.2010, Wisła, Poland, 43–46.

[18] Skruch P. (2011). A coverage metric to evaluate tests for continuous-time dynamic systems. *Central European Journal of Engineering*, **1**(2), 174–180.

[19] Skruch P., Panek M., Kowalczyk B. (2011). Model-based testing in embedded automotive systems. [in:] Zander-Nowicka J., Schieferdecker I., Mosterman P.J. (Eds.), *Model-Based Testing for Embedded Systems*. CRC Press (accepted for publication).

[20] Skeel R. (1992). Roundoff error and the Patriot missile. *Society for Industrial and Applied Mathematics (SIAM) News*, **25**(4), p. 11.

[21] National Institute of Standards & Technology, U.S. Department of Commerce: The economic impacts of inadequate infrastructure for software testing. Final Report, May 2002.

[22] The Institute of Electrical and Electronics Engineers, Inc.: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990. www.standards.ieee.org, 1990.

[23] Utting M., Legeard B. (2006). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, San Francisco.

[24] Whitesitt J.E. (1995). *Boolean Algebra and Its Applications*. Dover Publications, New York.

[25] Zander-Nowicka J., Schieferdecker I., Perez A.M. (2006). Automotive validation functions for online test evaluation of hybrid real-time systems. [in:] *Proceedings of the IEEE 41st Anniversary of the Systems Readiness Technology Conference (AutoTestCon 2006)*, September 2006, Anaheim, USA, 799–805.

[26] Zander-Nowicka J. (2009). *Model-based testing of embedded systems in the automotive domain*. PhD thesis, Technical University Berlin, Fraunhofer IRB Verlag.