Mathematical and Implementation Challenges Associated with Testing of the Dynamical Systems

Pawel Skruch

Mathematical and Implementation Challenges Associated with Testing of the Dynamical Systems

Pawel Skruch

AGH University of Science and Technology,
Faculty of Electrical Engineering, Automatics, Computer Science and Electronics,
Department of Automatics,
al. A. Mickiewicza 30/B1, 30–059 Krakow, Poland
pawel.skruch@agh.edu.pl

Abstract. The paper presents mathematical and implementation challenges associated with testing of embedded software systems with dynamic behavior. 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 with the help of abstract models that describe behavior of the software systems. The models are represented using the state space (or input/state/output) notation. An application example is given to illustrate theoretical analysis and mathematical formulation.

Keywords: software system; model-based testing; dynamical system

1 Introduction

Designing an embedded control system is a complex and error prone task. Within the last decades embedded systems have become increasingly sophisticated and their software content has grown rapidly. The increasing miniaturization of embedded control systems on the one hand and rising functional demands on the other hand require advanced and automated development and testing methodologies. In this context, model-based development (MBD) and model-based testing (MBT) approaches have the potential to facilitate the development of such systems under pressure of time-to-market constraints, quality assurance, and safety standards.

MBD is a process that provides the ability to graphically represent requirements, specification, and designs using domain-specific notations and simulate the resultant behavior for validation purposes. The code can be then generated from models, ranging from system skeletons to complete, deployable products. MBT is a related part that supports test generation from various kinds of models by application of a number of sophisticated methods.

Testing is the process of trying to discover every conceivable fault or weakness in a work product. The primary goal of the testing process is to found defects; the

secondary goal is to show the system's compliance to its requirements. Testing can show that defects are present, but cannot prove that there are no defects [9]. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness. Poorly tested systems may cost producers billions of dollars annually especially when defects are found by end users in production environments [7], [8], [13]. Barry Boehm's research analysis [4] 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 [10] estimated that software defects cost the U.S. economy $60 billion per year.

Exhaustive testing is impossible what means that testing everything (all combinations of inputs and preconditions) is not feasible expect for trivial cases. This is valid in particular for software systems with dynamic behavior. The dynamic systems are modeled by difference or differential equations and have usually infinitely many states. Testing dynamic aspects of such systems requires tests that utilize time continuous input signals and time continuous output signals (even when the system is digitally processed). The process of selecting just a few of the many possible scenarios to be tested is a difficult and challenging task and currently is most often based on qualitative best engineering judgment.

In this paper, testing problem as well as test artifacts for software systems with dynamic behavior are formulated using the mathematical formalism. The main results concern the concept of testing with a model as an oracle (section 2), a proposal for test notation (section 4), an implementation of a test comparator (section 5), a calculation of test coverage (section 6), and a selection of tests (sections 7). An example (section 8) is given to present a perspective on the applicability of the approach for industrial projects.

2 Concept of Testing with a Model as an Oracle

The model of a software system shall specify the system's behavior in a clear and unambiguous form. It can be used in computer simulations in an early phase of the development to validate the system concept, calibrate parameters, and optimize the system performance. In the next phase, the physical system is designed (i.e., hardware and software) that shall meet the requirements specified by the model. Testing process shall be considered as the last phase in the development process that allows verifying that the physical system behavior is identical to that observed during computer simulations. When the tests failed then the system needs to be redesigned. The physical system that is being tested for the correct operation is often referred to as the system under test (SUT).

The model fully represents the requirements therefore it can be used an oracle to assess if the algorithm implemented in the electronic control unit (ECU) being tested correctly implements the requirements. The term *test oracle* describes a source to determine expected results to compare with the actual result of the

SUT [1]. The approach of a validated model being used as an oracle (the block *Mathematical Model of the SUT* on figure 1) is very popular in industry and often applied. 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 $\boldsymbol{u}(\cdot)$ are applied to both the SUT and to the model. Next, the responses from the SUT $\boldsymbol{y}_{\mathrm{s}}(\cdot)$ and from the model $\boldsymbol{y}(\cdot)$ are compared by a test comparator to determine whether a test case has passed or failed.

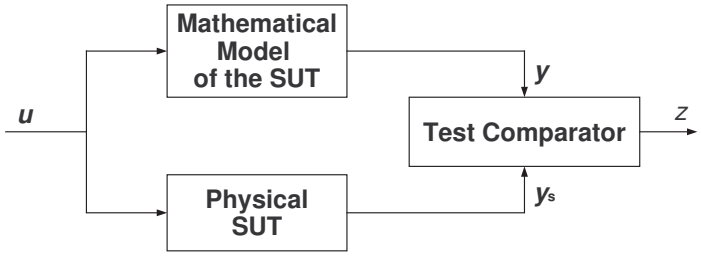

Fig. 1. Testing approach of a validated model being used as an oracle

3 Mathematical Model of the System Under Test

The state space (or input/state/output) representation provides a convenient way to model and analyze dynamical systems. The state space model consists of a set of input, output, and internal state variables that are expressed as vectors. The relationship between inputs, outputs, and internal states in a finite-dimensional, time-invariant, nonlinear system with continuous-time parameter can be specified by the following equations:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \ , \tag{1}$$

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

where $\boldsymbol{x}(t) \in X \subset \mathbb{R}^n$ refers to the internal state, $\boldsymbol{u}(t) \in U \subset \mathbb{R}^r$ refers to the input state, $\boldsymbol{y}(t) \in Y \subset \mathbb{R}^m$ refers to the output state, 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 internal state space, Y is the output state space, U is called the input state space, $\mathbb{R}^n$, $\mathbb{R}^m$, $\mathbb{R}^r$ are real vector spaces of column vectors, n, m, r are positive integers that determine numbers of internal state, output, and input variables, respectively.

The physical and implementation constraints imposed by computer system resources lead to the assumption that the spaces U, X, and Y shall be bounded. The assumption means that each space is contained in a ball of finite radius.

4 Test Notation

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 [15]. Adapting this definition to the state space modeling concept of the SUT (1), (2), a single test case $\mathrm{T}_{\mathrm{case}}^{(j)}$ can be defined as

$$\mathrm{T}_{\mathrm{case}}^{(j)} = \left\{ T^{(j)}, \boldsymbol{x}_0^{(j)}, \boldsymbol{u}^{(j)}(\cdot), \boldsymbol{y}^{(j)}(\cdot) \right\} , \tag{3}$$

in case of black-box testing [3], or

$$\mathrm{T}_{\mathrm{case}}^{(j)} = \left\{ T^{(j)}, \boldsymbol{x}_0^{(j)}, \boldsymbol{u}^{(j)}(\cdot), \boldsymbol{x}^{(j)}(\cdot), \boldsymbol{y}^{(j)}(\cdot) \right\} , \tag{4}$$

in case of gray-box testing [12]. Here, $\boldsymbol{u}^{(j)} : [0, T^{(j)}] \to \mathbb{R}^r$ 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 within the execution time window $[0, T^{(j)}]$ 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. 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 Test Comparator Implementation

The test comparator can be considered as a tool that implements a mechanism for determining whether a test has passed or failed [5]. In the concept, illustrated on figure 1, this tool compares the actual output $\boldsymbol{y}_{\mathrm{s}}(\cdot)$ produced by the SUT with the expected output $\boldsymbol{y}(\cdot)$ produced by the model. If the actual output is within a predefined tolerance range ϵ relative to the expected output, then the test is qualified as *pass* ($z = 0$, *system ok*), otherwise the test is qualified as *fail* ($z = 1$, *system error*). A possible practical realization of the comparison function z for a given test case $\mathrm{T}_{\mathrm{case}}^{(j)}$ is presented below:

$$z(\mathrm{T}_{\mathrm{case}}^{(j)}) = \begin{cases} 0 & \text{if } \forall_{t \in [0, T^{(j)}]} \left\| \boldsymbol{y}^{(j)}(t) - \boldsymbol{y_s}^{(j)}(t) \right\| < \epsilon \left\| \boldsymbol{y}^{(j)}(t) \right\| \\ 1 & \text{otherwise} . \end{cases} , \tag{5}$$

In the formula (5) the standard Euclidean norm $\| \cdot \|$ has been used to measure the distance between two points in the space $\mathbb{R}^m$.

6 Test Coverage Calculation

The degree to which a given test suite $\mathrm{T}_{\mathrm{suite}}$ addresses all specified requirements for a given system is determined by a test coverage measure [15]. The most obvious quantification of the system's behavior exercised by the test suite is computed by dividing the number of the system states explored by the test suite by the cardinality of the entire state space. However, the formula has limited usefulness for dynamical systems because the state space for such systems contains usually infinite number of states. In such situation, one of the possible ways out is to transform the internal state space X into another one $X_{\boldsymbol{h}}$ that contains countable number of elements.

The test coverage $C_{\boldsymbol{h}}$ of the test suite $\mathrm{T}_{\mathrm{suite}} = \{\mathrm{T}_{\mathrm{case}}^{(1)}, \mathrm{T}_{\mathrm{case}}^{(2)}, \ldots, \mathrm{T}_{\mathrm{case}}^{(N)}\}$ can be defined as follows [14]

$$C_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}}) = \frac{\left| \bigcup_{j=1}^{N} V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{case}}^{(j)}) \right|}{|X_{\boldsymbol{h}}|} , \tag{6}$$

where

$$X_{\boldsymbol{h}} = \{\boldsymbol{i} \in \mathbb{Z}^n : \exists_{\boldsymbol{x} \in X} : \boldsymbol{x} \in G_{\boldsymbol{h}}(\boldsymbol{i})\} \tag{7}$$

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

$$G_{\boldsymbol{h}}(\boldsymbol{i}) = \left\{ \boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{x} = [x_1\, x_2\, \ldots\, x_n]^{\mathrm{T}} ,\ \left\lfloor \frac{x_k}{h_k} \right\rfloor = i_k,\ k = 1, 2, \ldots, n \right\} \tag{8}$$

denotes a partition with the size $\boldsymbol{h}$ in the space $\mathbb{R}^n$, $\left\lfloor \frac{x_k}{h_k} \right\rfloor$ is the largest integer not greater than $\frac{x_k}{h_k}$,

$$V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{case}}^{(j)}) = \left\{ \boldsymbol{i} \in X_{\boldsymbol{h}} : \exists_{t \in [0, T^{(j)}]} : \boldsymbol{x}^{(j)}(t) \in G_{\boldsymbol{h}}(\boldsymbol{i}) \right\} \tag{9}$$

is a set of states of the transformed internal state space covered by the test case $\mathrm{T}_{\mathrm{case}}^{(j)}$. It should be noticed that the sum

$$V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}}) = \bigcup_{j=1}^{N} V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{case}}^{(j)}) \tag{10}$$

will contain the information about the internal states covered by the test suite $\mathrm{T}_{\mathrm{suite}}$.

The proposed test coverage measure is defined using a partition (or discretization) of the system internal state 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 system state during a test.

7 Conformance Test Selection Method

The section presents a proposal of the algorithm for generating test cases. The general principle of the algorithm is to create input functions $\boldsymbol{u}(\cdot)$ for which the system trajectories $\boldsymbol{x}(\cdot)$ cross every element $G_{\boldsymbol{h}}(\boldsymbol{i})$ of the space $X_{\boldsymbol{h}}$. The selection and completeness of test cases is quantified by the coverage metric (6). Test cases are selected to check that the functional specification (here in the form of the mathematical model) is correctly implemented, which is variously referred to in the literature as conformance testing [5], correctness testing [6], or functional testing [15].

Algorithm 1 Conformance test selection method

1: $\boldsymbol{h} = [h_1\, h_2\, \ldots\, h_n]^{\mathrm{T}}$, $h_1, h_2, \ldots, h_n > 0$, $\delta \in (0, 1]$, $T > 0$
2: $\mathrm{T}_{\mathrm{suite}} := \emptyset$, $V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}}) := \emptyset$, $C_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}}) = 0$, $j := 0$
3: **while** $C_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}}) \leq \delta$ **do**
4: Find $\boldsymbol{x}_a \in G_{\boldsymbol{h}}(\boldsymbol{i}_a)$, $\boldsymbol{x}_b \in G_{\boldsymbol{h}}(\boldsymbol{i}_b)$ where $\boldsymbol{i}_b \in X_{\boldsymbol{h}} \backslash V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}})$
5: Calculate the control function $\boldsymbol{u}^*(\cdot)$ that steers the system from the initial state $\boldsymbol{x}(0) = \boldsymbol{x}_a$ to the final state $\boldsymbol{x}(T) = \boldsymbol{x}_b$ at finite time T
6: Calculate the trajectory $\boldsymbol{x}^*(\cdot)$ and output function $\boldsymbol{y}^*(\cdot)$
7: $j := j + 1$
8: $\mathrm{T}^{(j)}_{\mathrm{case}} := \{T^{(j)}, \boldsymbol{x}^{(j)}_0, \boldsymbol{u}^{(j)}(\cdot), \boldsymbol{x}^{(j)}(\cdot), \boldsymbol{y}^{(j)}(\cdot)\}$, where $T^{(j)} := T$, $\boldsymbol{x}^{(j)}_0 := \boldsymbol{x}_a$, $\boldsymbol{u}^{(j)}(\cdot) := \boldsymbol{u}^*(\cdot)$, $\boldsymbol{x}^{(j)}(\cdot) := \boldsymbol{x}^*(\cdot)$, $\boldsymbol{y}^{(j)}(\cdot) := \boldsymbol{y}^*(\cdot)$
9: $\mathrm{T}_{\mathrm{suite}} := \mathrm{T}_{\mathrm{suite}} \cup \mathrm{T}^{(j)}_{\mathrm{case}}$
10: Calculate $V_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}})$ and $C_{\boldsymbol{h}}(\mathrm{T}_{\mathrm{suite}})$
11: **end while**

8 Embedded PID Controller Example

An embedded PID controller is a system that can be considered as a combination of computer hardware and software designed to perform a dedicated control function. The PID controller works in a closed-loop system (figure 2) and attempts to minimize the error $e(t)$ by adjusting the control input $s(t)$. The error is calculated as the difference between a measured process output $v(t)$ and a desired set point $v_{\mathrm{sp}}(t)$. The control signal is a result of the following calculation

$$s(t) = K\left(e(t) + \frac{1}{T_{\mathrm{i}}}\int_0^t e(\tau)\,\mathrm{d}\tau + T_{\mathrm{d}}\frac{\mathrm{d}e(t)}{\mathrm{d}t}\right), \tag{11}$$

where $K = 3.6$ is proportional gain, $T_{\mathrm{i}} = 1.81\,[\mathrm{s}]$ is integral time, $T_{\mathrm{d}} = 0.45\,[\mathrm{s}]$ is derivative time. The control signal is thus a sum of three terms: the P-term (which is proportional to the error), the I-term (which is proportional to the integral of the error), and D-term (which is proportional to the derivative of the error). The parameters K, T_{i}, and T_{d} can be obtained using the Ziegler-Nichols

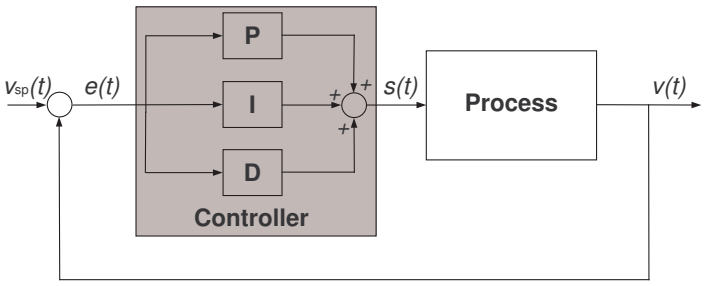

Fig. 2. A block diagram of the closed-loop system with the PID controller

algorithm [16]. The model of the process to be controlled has been omitted in the example for the purpose of clarity and easy readability.

The algorithm 1 can be used to generate a set of conformance test cases. Before its execution, the equation (11) needs to be rewritten to the form

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}u(t), \quad \boldsymbol{x}(0) = \boldsymbol{0}\,, \tag{12}$$

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t)\,, \tag{13}$$

where $\boldsymbol{x}(t) = [x_1(t)\ x_2(t)]^{\mathrm{T}}$, $x_1(t) = \int_0^t e(\tau)\,\mathrm{d}\tau$, $x_2(t) = \dot{x}_1(t)$,

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 \\ -(T_{\mathrm{i}}T_{\mathrm{d}})^{-1} & -T_{\mathrm{d}}^{-1} \end{bmatrix}\,, \quad \boldsymbol{B} = \begin{bmatrix} 0 \\ (KT_{\mathrm{d}})^{-1} \end{bmatrix}\,, \quad \boldsymbol{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\,. \tag{14}$$

Then, the algorithm has been implemented and executed with the following parameters: $\boldsymbol{h} = [0.3,\ 0.2]^{\mathrm{T}}$ (size of the partition), $\delta = 0.7$ (acceptable coverage level), $T = 20\,[\mathrm{s}]$ (test execution time), $-1.5 \leq x_1(t) < 1.5$, $-1 \leq x_2(t) < 1$ (system implementation constraints). The test suite that guarantees the coverage level higher than δ consists of 10 test cases. The elements of these test cases are detailed in table 1, figures 3 and 4.

9 Conclusions

In spite of continuing research on test approaches for continuous and mixed discrete-continuous systems, 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. The model-based approach presented in the paper looks promising. The functional model of the system under test can be used as an oracle providing the capabilities to assess the results of test cases in an automatic way and also in test generation algorithms. Additional aspects, such as test notation, implementation of a test comparator, and coverage analysis have been discussed in the paper in order to have complete set of tools and mathematical methods for testing software systems with dynamic behavior. The example has been used to validate the concept and to have a perspective on its applicability for industrial projects.

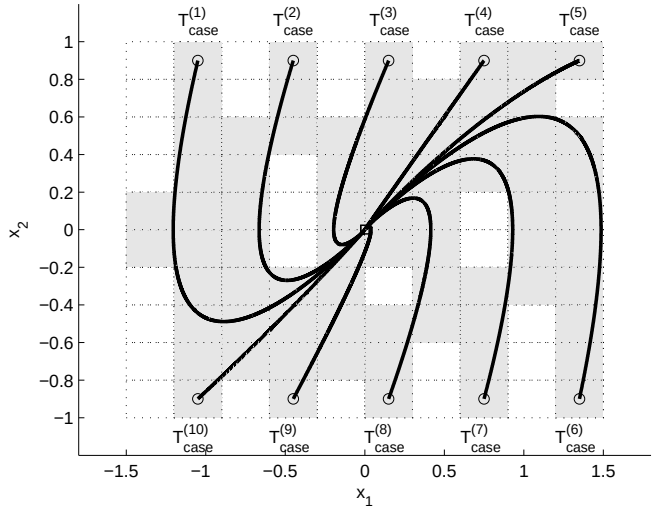

Fig. 3. Trajectories $\boldsymbol{x}^{(j)}$, $j = 1, 2, \ldots, 10$ and elements (*gray rectangles*) of the transformed state space covered by the test cases $T_{\text{case}}^{(j)}$. The trajectories start in □ and end in ∘.

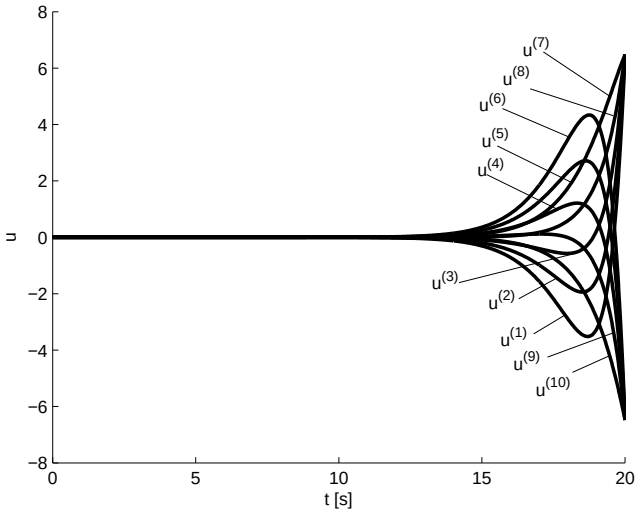

Fig. 4. Illustration of the input functions $u^{(j)}(\cdot)$, $j = 1, 2, \ldots, 10$ belonging to the test cases $\mathrm{T}_{\text{case}}^{(j)}$

Table 1. An example test report for the test suite $\mathrm{T}_{\mathrm{suite}} = \{\mathrm{T}^{(1)}_{\mathrm{case}}, \mathrm{T}^{(2)}_{\mathrm{case}}, \ldots, \mathrm{T}^{(10)}_{\mathrm{case}}\}$ that guarantees the coverage level $C_{\boldsymbol{h}} > \delta$, where $\boldsymbol{h} = [0.3, 0.2]^{\mathrm{T}}$, $\delta = 0.7$ (70%). The notation used in the last column means $\mathrm{T}^{(j)}_{\mathrm{suite}} = \{\mathrm{T}^{(1)}_{\mathrm{case}}, \mathrm{T}^{(2)}_{\mathrm{case}}, \ldots, \mathrm{T}^{(j)}_{\mathrm{case}}\}$.

j	$T^{(j)}$[s]	$\boldsymbol{x}_0^{(j)\mathrm{T}}$	$u^{(j)}(\cdot)$	$\boldsymbol{x}^{(j)}(\cdot)$	$\boldsymbol{y}^{(j)}(\cdot)$	$C_{\boldsymbol{h}}(\mathrm{T}^{(j)}_{\mathrm{case}})$	$C_{\boldsymbol{h}}(\mathrm{T}^{(j)}_{\mathrm{suite}})$
1	20	[0, 0]	fig. 4	fig. 3		0.16	0.16
2	20	[0, 0]	fig. 4	fig. 3		0.12	0.24
3	20	[0, 0]	fig. 4	fig. 3		0.08	0.30
4	20	[0, 0]	fig. 4	fig. 3		0.07	0.36
5	20	[0, 0]	fig. 4	fig. 3		0.09	0.40
6	20	[0, 0]	fig. 4	fig. 3		0.14	0.49
7	20	[0, 0]	fig. 4	fig. 3		0.12	0.58
8	20	[0, 0]	fig. 4	fig. 3		0.08	0.65
9	20	[0, 0]	fig. 4	fig. 3		0.08	0.70
10	20	[0, 0]	fig. 4	fig. 3		0.09	0.73

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

References

1. Adrion, W., Brandstad, J., Cherniabsky, J.: Validation, Verification and Testing of Computer Software. Computing Surveys 14, 159–192 (1982)
2. Beizer, B.: Software Testing Techniques, 2nd ed. Van Nostrand Reinhold, Boston, USA (1990)
3. Beizer, B.: Black-Box Testing. Techniques for Functional Testing of Software and Systems. John Willey & Sons, New York, USA (1995)
4. Boehm, B.: Software Engineering Economics. Prentice Hall, Englewood Cliffs, USA (1981)
5. International Software Testing Qualifications Board (ISTQB): Standard Glossary of Terms Used in Software Testing, Version 2.1 (2010), `http://www.astqb.org`
6. Kaner, C., Faulk, J., Nguyen, H.Q.: Testing Computer Software, 2nd ed. John Willey & Sons, New York, USA (1995)
7. Leveson, N.G., Turner, C.S.: An Investigation of the Therac-25 Accidents. IEEE Computer 27, 18–41 (1993)
8. Lions, J.L.: ARIANE 5. Flight 501 Failure. Ariane 501 Inquiry Board Report. Technical report, Paris, France (1996)
9. Myers, G.: The Art of Software Testing, 2nd ed. John Willey & Sons, New York, USA (2004)
10. National Institute of Standards & Technology, U.S. Department of Commerce: The Physiology of the Grid: The Economic Impacts of Inadequate Infrastructure for Software Testing, Final Report. Technical report, North Carolina, USA (2002)
11. Nelder, J.A., Mead, R.: A Simplex Method for Function Minimization. The Computer Journal 7, 308–313 (1965)
12. Patton, R.: Software Testing, 2nd ed. Sams, Indianapolis, USA (2005) +
13. Skeel, R.: Roundoff Error and the Patriot Missile. Society for Industrial and Applied Mathematics (SIAM) News 25, 11 (1992)

14. Skruch, P.: A Coverage Metric to Evaluate Tests for Continuous-Time Dynamic Systems. Central European Journal of Engineering 1, 174–180 (2011)
15. The Institute of Electrical and Electronics Engineers, Inc.: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990 (1990), `http://www.standards.ieee.org`
16. Ziegler, J., Nichols, N.: Optimum Settings for Automatic Controllers. Transactions of ASME 64, 759-768 (1942)