11

TEST TEMPLATES FOR TEST GENERATION

Marco Hollenberg
Philips Research Laboratories Eindhoven
hollenbe@natlab.research.philips.com

Abstract We describe the use of *test templates* in test generation from deterministic Finite State Machines. A test template is an expression in a formal language that describes a test or group of tests. Such templates can be used to guide the test generation process.

Keywords: Conformance testing, TTCN, Finite State Machines.

1. INTRODUCTION

Within Philips, we have developed PHACT, the Philips Automated Conformance Tester ([3]). From deterministic Finite State Machines (FSMs) TTCN ([4]) test suites are generated. This is powered by the Conformance Kit ([1]), which provides a number of fixed strategies for test generation, principal amongst these a UIO-method ([5]), referred to as the partitioned tour method. PHACT also has test execution environments for different platforms (pSOS, Windows) such that these suites can be executed. For this we of course need to write a translation function, linking the abstract events of the TTCN test suite to concrete events that the System Under Test (SUT) understands.

This paper is the result of research targeting the test generation module of PHACT (i.e, the Conformance Kit). We were dissatisfied with the fact that only a fixed number of strategies were given to us. The ability to guide the generation process was considered desirable. The need for this occurs both when one deals with large and with relatively small specifications.

If a specification is large the strategies given are too expensive. Calculating UIO-sequences for thousands of states is not always feasible. Even if it was, executing the resulting test suite could be too expensive. In large specifications, we would like to steer the test generation process so that it at least covers certain areas of the specification that we find interesting.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35567-2_25

G. Csopaki et al. (eds.), *Testing of Communicating Systems*

It may also occur that your specification is rather small. We have encountered the need for small specifications in a number of our projects. This does not necessarily mean that the SUTs have a small statespace. Typically, we do not model the behaviour of the entire system in one specification. Instead, we produce a number of specifications, each dealing with a different aspects of the system. This is because our SUTs are not always purely protocol oriented, but their behaviour may also be governed by a quickly changing environment, such as an audio/video bitstream, which may not be under adequate control of the tester. In such cases it may be necessary to have different specifications, taking into account different environments. The result is that we get a number of possibly quite small specifications.

If we use the strategies supplied by the Conformance Kit for test generation, this may not yield a very large test suite. In PHACT, once a test suite generated from a particular FSM can be executed (the translation function between abstract and concrete events has been written, the test environment has been set up), everything is in place to execute other test suites that are generated from the same FSM. Thus, if *more* tests could be generated, above and beyond the ones generated using the supplied strategies, these could be executed almost for free. A greater number of tests provides the possibility of more exhaustive testing which will yield a better coverage and hence a greater confidence in correctness of the implementation. Of course, we do not just want test suites with randomly generated tests: we want some control, we want to guide the generation of these extra test suites.

To provide more guidance to the test generation process in PHACT, we came up with the notion of *test templates*. A test template is an expression in a formal language (akin to that of regular expressions) that allows you to state the structure of a test, what building blocks should go into the test and how they may be combined. The idea is to generate from such a test template a test suite made up out of tests that satisfy the template. In this way, we can specify a greater number of strategies than we had available before, and guide the generation process towards test suites that we find desirable.

2. TRACES AND TESTS

In this section we provide the background for the definition of test templates. We begin by defining what kind of specifications we consider and what constitutes a test whether an SUT conforms to the specification.

Definition 2..1
A *Finite State Machine* (FSM) is a tuple $\mathcal{F} = (S, L_I, L_O, \rightarrow, s_0)$, where:

- S is a nonempty finite set of *states*;
- L_I is a finite set of *input actions*;

- L_O is a finite set of *output actions*;
- $\rightarrow \subseteq S \times L_I \times L_O \times S$ is the set of *transitions*.
- $s_0 \in S$ is the *initial state*.

□

There is no requirement that L_I is disjoint from L_O, as sometimes happens in the definition of FSMs, although requiring such a disjointness would not change anything in this paper.

Let us fix some notation that is convenient when discussing FSMs:

- $s \xrightarrow{a?/b!} t$ denotes: $(s, a, b, t) \in \rightarrow$.
- $s \xrightarrow{a_1?/b_1!\ldots a_n?/b_n} t$ denotes: there are $s_1, \ldots, s_n \in S$ such that $s \xrightarrow{a_1?/b_1!} s_1 \ldots \xrightarrow{a_n?/b_n!} s_n = t$.
- If in the notation $s \xrightarrow{a_1?/b_1!\ldots a_n?/b_n!} t$ we omit the outputs $b_1, \ldots, b_n$ or the state t we mean that these can be found such that the statement becomes true. For instance: $s \xrightarrow{a_1?\ldots a_n?}$ means that there are outputs $b_1, \ldots, b_n \in L_O$ and there is a state $t \in S$ such that: $s \xrightarrow{a_1?/b_1!\ldots a_n?/b_n!} t$.

Next, we list some properties of FSMs. The FSMs that we will consider in this paper all have these properties.

- An FSM is *deterministic* if for every $s \in S$ and every $a \in L_I$ there is at most one pair (b, t) such that $s \xrightarrow{a?/b!} t$.
- An FSM is *synchronizing* if there is a sequence $a_1, \ldots, a_n \in L_I$ such that $s \xrightarrow{a_1?\ldots a_n?} s_0$ for every state $s \in S$. This means that we can use that sequence to take us from any state to the initial state. The sequence $a_1 \ldots a_n$ is the called a synchronizing sequence.
- An FSM is *connected* if for every $s \in S$ there is some sequence $a_1 \ldots a_n \in L_I$ such that $s_0 \xrightarrow{a_1?\ldots a_n?} s$. I.e., every state can be reached from the initial state.

Throughout this paper we only consider deterministic synchronizing connected FSMs.

Definition 2..2

A *trace* through an FSM $\mathcal{F}$ is a pair $(s, a_1 \ldots a_n)$ with each $a_i \in L_I$ such that $s \xrightarrow{a_1?\ldots a_n?}$.

Because we assume FSMs are deterministic we can define the *output sequence* of a trace $(s, a_1 \ldots a_n)$ as the unique sequence $b_1, \ldots, b_n \in L_O$ such that $s \xrightarrow{a_1?/b_1! \ldots a_n?/b_n!}$ (such a sequence must exist, if $(s, a_1 \ldots a_n)$ is in fact a trace as defined above).

□

A *test* of a deterministic FSM is defined to be a trace that begins in the initial state of the FSM. An implementation passes a test $(s_0, a_1 \ldots a_n)$ with output sequence $b_1, \ldots, b_n$ if after bringing the system into its initial state (by means of a synchronizing sequence) we successively supply the stimuli $a_1, \ldots, a_n$, it responds with, respectively, $b_1, \ldots, b_n$. Otherwise it fails the test. From a test specified as a trace it is possible to create a TTCN test case. Section 5. contains an example of such a test case (table 1).

3. TEST TEMPLATES

Now that everything is in place, it is time to define the notion of test templates. As tradition dictates we first define the syntax, then the semantics.

3.1 SYNTAX

Given a set of S of FSM-states the set of *guards over* S is defined as follows:

$$\phi ::= s \mid \top \mid \phi \vee \phi \mid \neg\phi$$

where $s \in S$. In other words: the set of guards over S is simply the set of boolean expressions over S. A guard will be interpreted (formally defined in the next section) as a set of FSM-states.

From the same set S of states and a set L_I of input-actions we can define the set of *test templates over* S *and* L_I as follows:

$$\pi ::= a \mid . \mid [\phi] \mid \text{TS} \mid \text{SIOS} \mid \pi + \pi \mid \pi;\pi \mid \pi^*$$

where $a \in L_I$ and ϕ is a guard over S.

Test templates are in essence regular expressions. They will be interpreted as sets of traces. An example of a test template is:

$$\text{TS}; x; ([s \vee t]; y)^*; [s \vee t] \tag{1}$$

A trace satisfies this template if it starts with a *transfer sequence* to an arbitrary state (a sequence of inputs that brings us from the intial state to this arbitrary state, calculated by some efficient function), then does an x-step so that we end up in either state s or state t, and then does an arbitrary number of y-steps so that the states s or t are not left. This could be used to guide

the test generation process towards tests that perform y-actions in states s or t, preceded by an x-action. All of this will be formally defined in the next section.

If an FSM $\mathcal{F}$ is clear from the context, a test template is understood to be a test template over its set of states and its set of input-actions and similarly for guards.

3.2 SEMANTICS

Test templates are to be interpreted as sets of traces. Of ultimate interest are only the *tests* among these: the ones that start in the initial state. We give a formal definition in this section. Assume fixed an FSM $\mathcal{F} = (S, L_I, L_O, \rightarrow, s_0)$.

Guards are interpreted as subsets of S. If ϕ is a guard, its interpretation $[\![\phi]\!]$ is defined as follows:

$$\begin{array}{lcll} [\![s]\!] & = & \{s\} & \text{if } s \in S \\ [\![\top]\!] & = & S & \\ [\![\phi \vee \psi]\!] & = & [\![\phi]\!] \cup [\![\psi]\!] & \\ [\![\neg\phi]\!] & = & S \setminus [\![\phi]\!] & \end{array}$$

Test templates are interpreted as sets of traces. They are build up from guards, input-actions and two special building blocks: transfer sequences and Simple Input-Output Sequences (SIOSs). These need to be defined before we can proceed with the semantics of test templates.

A *transfer sequence* to a state $s \in S$ is a trace $(s_0, a_1 \ldots a_n)$ such that $s_0 \xrightarrow{a_1?\ldots a_n?} s$. We assume present a function `makets` that, for every $s \in S$ produces a transfer sequence to s. Such a function can exist because we assume FSMs to be connected.

A SIOS for a state s is a trace $(s, a_1 \ldots a_n)$ such that for every $t \in S$: $(t, a_1 \ldots a_n)$ is a trace and its output sequence differs from that of $(s, a_1 \ldots a_n)$. A SIOS need not exist for every state. SIOSs can be used to gain confidence in the idea that at some point of a test the SUT has reached a particular state and not some other. We assume present a function `makesios` that for any $s \in S$ produces a SIOS for s if it exists, and yields the trace (s, ε) otherwise (where ε is the empty sequence).

Now we can define the semantics of test templates:

- $[\![a]\!] = \{(s, a) \mid s \in S \;\&\; s \xrightarrow{a}\}$.
- $[\![.]\!] = \{(s, a) \mid s \in S \;\&\; a \in L_I \;\&\; s \xrightarrow{a}\}$.
- $[\![[\phi]]\!] = \{(s, \varepsilon) \mid s \in [\![\phi]\!]\}$.
- $[\![\text{TS}]\!] = \{\texttt{makets}(s) \mid s \in S\}$.
- $[\![\text{SIOS}]\!] = \{\texttt{makesios}(s) \mid s \in S\}$.

- $[\![\pi_1 + \pi_2]\!] = [\![\pi_1]\!] \cup [\![\pi_2]\!]$.
- $[\![\pi_1; \pi_2]\!] = [\![\pi_1]\!] \circ [\![\pi_2]\!]$ where $\circ$ is defined on sets of traces as follows: $T_1 \circ T_2 =$

$$\left\{ (s, a_1 \ldots a_n a_{n+1} \ldots a_{n+m}) \mid \begin{array}{l} (s, a_1, \ldots, a_n) \in T_1 \ \& \\ \exists t \in S. \left(\begin{array}{l} s \xrightarrow{a_1?\ldots a_n?} t \ \& \\ (t, a_{n+1} \ldots a_{n+m}) \in T_2 \end{array} \right) \end{array} \right\}$$

- Using the above notion of composition we can also define trace *iteration*. If T is a set of traces we define:

$$\begin{array}{lcl} T^0 & = & \{(s, \varepsilon) \mid s \in S\} \\ T^{n+1} & = & T^n \circ T \\ T^* & = & \bigcup_{n \geq 0} T^n \end{array}$$

  Now we can define: $[\![\pi^*]\!] = [\![\pi]\!]^*$.

Consider again the test template 1. The interpretation of this will consist of all traces $(s_0, a_1 \ldots a_n x y_1 \ldots y_m)$ such that for some state u, $\texttt{makets}(u) = (s_0, a_1 \ldots a_n)$ and for some states $v_0, \ldots, v_m \in \{s, t\}$, $s_0 \xrightarrow{a_1?\ldots a_n?} u \xrightarrow{x?} v_0 \xrightarrow{y_1?} v_1 \ldots \xrightarrow{y_m?} v_m$ and $y_1 = \ldots = y_m = y$. It depends completely on the FSM if such traces exist or not.

4. THE RESULTING TEST SUITE

If we have an FSM and a test template π, what tests do we want in our test suite? We can not always include all tests that satisfy π. For a start, if π uses iteration, we would get an infinite suite. But even if this is not the case, the suite might get too large. A simple solution is to let the user specify a range of lengths of tests $[n, m]$, where $0 \leq n \leq m$. The length of a trace $(s, a_1 \ldots a_n)$ is the number of inputs in the trace: n in this case.

The test suite can be pruned some by removing redundancy. A test that tests what happens if we supply an input a_1 and then an input a_2 (from the initial state) subsumes a test that only tests a_1. Thus *prefixes* of tests can be removed from the suite.

If it is not always feasible for a test generation tool to generate a test suite containing all tests satisfying the template, even if we remove all redundant tests, we need to define what test suites a tool is allowed to generate. We give such a definition here:

Definition 4..1
Assume given a synchronizing, connected, deterministic FSM $\mathcal{F}$. Let π be a test template, and let $[n, m]$ be the specified range ($0 \leq n \leq m$). Then a test suite T *satisfying* π and $[n, m]$ is a set such that:

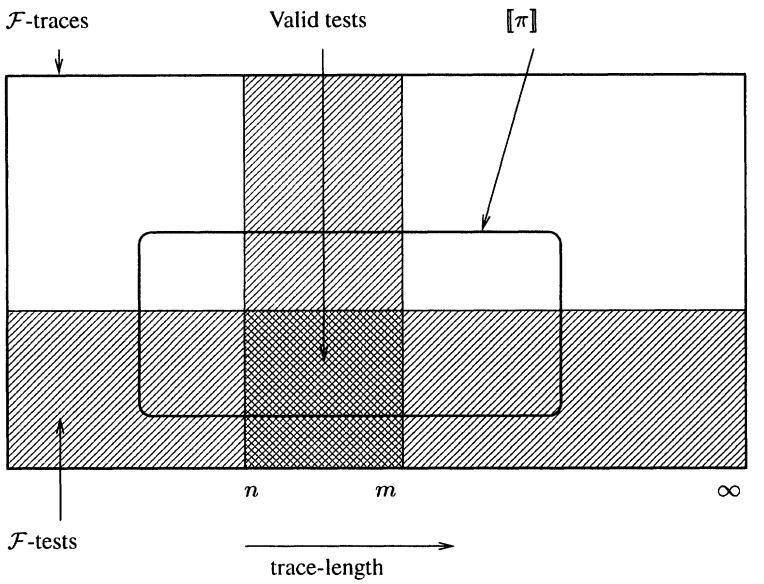

Figure 1 Tests generated from FSM, test template and specified range

- T satisfies the test template π: $T \subseteq [\![\pi]\!]$.
- T contains only tests: if $(s, a_1 \ldots a_k) \in T$ then $s = s_0$.
- T satisfies the length-requirement: if $(s_0, a_1 \ldots a_k) \in T$ then $n \leq k \leq m$.
- T contains no redundancies: if $(s_0, a_1 \ldots a_k) \in T$ then for no $l < k$: $(s_0, a_1 \ldots a_l) \in T$. A test subsumes all of its proper prefixes.

□

There can be many test suites that satisfy the same template and range. Some of these will be less informative (i.e. test less) than others. This notion is easy to define. Read $\sqsubseteq$ as "is less informative than, or just as informative". First $(s_0, a_1 \ldots a_n) \sqsubseteq (s_0, c_1 \ldots c_m)$ iff $a_1 \ldots a_n$ is a prefix of $c_1 \ldots c_m$ (or they are equal). Then for two test suites T_1 and T_2: $T_1 \sqsubseteq T_2$ iff for each $t \in T_1$ there is a $u \in T_2$ such that $t \sqsubseteq u$. Now it is easy to see that for each test template and each range there is a finite test suite that satisfies the above requirements and is maximal with respect to $\sqsubseteq$ among all those that satisfy the requirements. Such a test suite we call *maximal* with respect to the template and the range.

Figure 1 depicts where the tests that go into satisfying a template and a range are located.

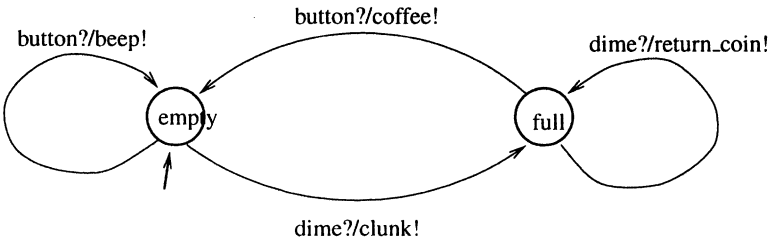

Figure 2 A coffee machine FSM

We have written a Unix prototype tool, `tp2ttcn`, that implements the ideas presented so far. From an FSM, a test template, a range, a specified maximum number of tests it produces a TTCN test suite that satisfies the template and the range. We will not go into the algorithm of the tool here. Suffice it to say the test template is used to build a tree whose nodes represent tests and where some nodes are labelled to indicate that they represent tests that satisfy the template.

5. AN EXAMPLE

In this section we work out a few examples of test templates, to get a better feel. Our running example is the FSM in figure 2. It is easily verified to be deterministic, synchronizing and connected. The synchronizing sequence is `button`. The FSM represents a simple coffee machine. Coffee costs a dime. If a dime has been inserted a button can be pressed to obtain coffee. Pressing the button too soon produces a warning and inserting too much money causes the excess to be returned to you.

A decent test template would be the *partitioned tour*:

TS; .; SIOS

This should produce a test suite such that each test case starts with a transfer sequence to some state, proceeds with some arbitrary step and finally does a SIOS to check that it really is in the state it expects to be in.

Suppose that the functions `makets` and `makesios` are defined as follows:

$$\texttt{makets}(s) = \begin{cases} (\texttt{empty}, \varepsilon) & \text{if } s = \texttt{empty} \\ (\texttt{empty}, \texttt{dime}) & \text{if } s = \texttt{full} \end{cases}$$

$$\texttt{makesios}(s) = (s, \texttt{dime})$$

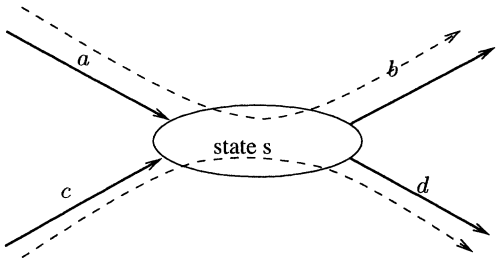

Figure 3 A traversal that does not test all pairs of transitions

Then $[\![\text{TS}; .; \text{SIOS}]\!] =$

$$\left\{ \begin{array}{l} (\texttt{empty}, \texttt{dime} \cdot \texttt{dime}), \\ (\texttt{empty}, \texttt{button} \cdot \texttt{dime}), \\ (\texttt{empty}, \texttt{dime} \cdot \texttt{dime} \cdot \texttt{dime}), \\ (\texttt{empty}, \texttt{dime} \cdot \texttt{button} \cdot \texttt{dime}) \end{array} \right\}$$

Because the template started with TS it is no coincidence that this set contains only tests: every trace that satisfies TS starts in the initial state. The maximal test suite for this template and a range of, say $[0, 3]$ consists of $[\![\text{TS}; .; \text{SIOS}]\!]$ without $(\texttt{empty}, \texttt{dime} \cdot \texttt{dime})$, which is redundant because of the presence of $(\texttt{empty}, \texttt{dime} \cdot \texttt{dime} \cdot \texttt{dime})$. The tool creates TTCN from such traces. A module of the Conformance Kit is used for this. Table 1 contains a few of the tables of the resulting TTCN, as generated by the Conformance Kit.

The partitioned tour happens to be one of the strategies that the Conformance Kit supplies. For us, it is just a template among others though. The partitioned tour produces tests that visit all transitions at least once. What if we are worried about *combinations* of transitions? Consider the partial state machine presented in figure 3. The dotted path show a traversal of the state machine by a test suite. All transitions are covered by the transitions. But note that we do not find errors in this way that occur when an a-transition is followed by a d-transition, or a c-transition is followed by a b-transition.

In the test template language we can specify an alternative to the partitioned tour that tests all transitions and all pairs of transitions:

$$\text{TS}; (. + (.; .)); \text{SIOS}$$

The maximal test suite for this template (taking a sufficiently broad range) contains six tests, even though the interpretation of the template (without removing redundancy) contains ten. If larger examples are considered the amount of redundancy could be a real bottleneck to test generation from templates, if we did not remove it at its source. The tool `tp2ttcn` does precisely that.

Table 1 TTCN tables

Test Case Dynamic Behaviour			
Test Case Name:	test_1		
Test Group:	coffeeMachine/coffee/		
Purpose:	Elementary sequence "test_1" (1) from state "empty"		
Defaults Reference:	general_default		
Nr	**Behaviour Description**	**CRef**	**Verdict**
1	+ ss		
2	pco ! button		
3	pco ? beep		
4	pco ! dime		
5	pco ? clunk		PASS
Detailed Comments:			
	line 1: *force IUT in start state "empty"*		

Test Step Dynamic Behaviour			
Test Step Name:	ss		
Test Group:	coffeeMachine/steps/coffee/		
Objective:	Synchronize to start state "empty"		
Defaults Reference:	general_default		
Nr	**Behaviour Description**	**CRef**	**Verdict**
1	pco ! button		
2	**START** no_output_timer		
3	+ gobble_one		
4	CANCEL no_output_timer		
	gobble_one		
5	pco **? OTHERWISE**		
6	**? TIMEOUT** no_output_timer		

Default Dynamic Behaviour			
Default Name:	general_default		
Test Group:	coffeeMachine/defaults/coffee/		
Objective:	On any other event: fail		
Nr	**Behaviour Description**	**CRef**	**Verdict**
1	pco **? OTHERWISE**		FAIL

Our final example is of a template with an infinity of tests that satisfy it. It is here where the need for specifying a range is clear. Suppose we wanted to create a large test suite containing tests that check whether the coffee machine does not give out free coffee, i.e. bring it to the `empty` state and press the button. A suitable test template for this purpose would be

```
.*;[empty];button
```

We could now say that we want all tests that satisfy this template but have a length ≤ 10: this would produce a maximal test suite of 2^9 test cases (if we remove redundancy). Note that such a test suite is no longer purely functional: it borders on a stress test.

6. CONCLUSIONS AND FURTHER RESEARCH

This paper has described test templates. They are useful for guiding the test generation process in relatively small specifications, where the requirement that every transition is covered by a test is easily met, but one still wants more. For large specifications they are also useful: covering every transition may not be feasible so one wants to guide the test generation process towards a test suite that covers all the *interesting* parts (whatever those may be).

We conclude with a list of possible extensions. First, our tool uses FSMs as its specification language. One could consider using slightly more advanced languages. For instance: EFSMs (Extended FSMs), FSMs extended with variables. One would then need to extend the test template language as well. An example of an extended test template would be:

$$\text{TS}; [\texttt{value} = 0 \vee \texttt{value} = 50 \vee \texttt{value} = 100]; .; \text{SIOS}$$

which denotes a partitioned tour that only considers steps from states where the `value`-variable has a boundary value (0 or 100) or a chosen medium value (50).

In our test template language we consider two functions, one for calculating transfer sequences, one for calculating SIOSs. This was done because these are available in the Conformance Kit. One could of course consider other functions, perhaps to implement other UIO-strategies, and add the corresponding building blocks to the language.

A final extension would be to consider *nondeterministic* FSMs. This would require a major overhaul of the above work. In nondeterministic FSMs, tests can no longer be equated with lists of input sequences. It makes much more sense to equate tests with *trees* labeled with both inputs and outputs. When tests are traces, it makes sense to choose a regular expression-like language for test templates. When tests are trees, this is no longer a good choice. A modal or temporal language, such as CTL ([2]), is more suitable. A drawback for us would be that the Conformance Kit could not be reused: this tool does not support nondeterministic FSMs.

References

[1] S.P. van de Burgt, J. Kroon, E. Kwast, H.J. Wilts. The RNL Conformance Kit. In: J. de Meer, L. Mackert and W. Effelsberg, eds., *Proc. of the 2nd*

International Workshop on Protocol Test Systems, pp. 279-294. North-Holland, 1989.

[2] E.M. Clarke, E.A. Emerson, A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. In: *ACM Transactions in Programming Languages and Systems*, Vol. 8, No. 2, 1986, pp. 244-263.

[3] L.M.G. Feijs, F.A.C. Meijs, J.R. Moonen, J.J. van Wamel. Conformance Testing of a Multimedia System Using PHACT. In: A. Petrenko, N. Yevtushenko, *Testing of Communicating Systems*, Proceedings of IWTCS'98, pp. 193–210, Kluwer, 1998.

[4] ISO. *Information Technology – Open Systems Interconnection, Conformance Testing Methodology and Framework* – Part 3: *Tree and Tabular Combined Notation*. International Standard IS)/IEC 9646-3, 1991.

[5] H. Ural. Formal methods for test sequence generation. In: *Computer Communications*, Vol 15, No. 5, 1992, pp. 311-325.