Integration of Functional and Timed Testing of Real-time and Concurrent Systems

Victor V. Kuliamin, Alexander K. Petrenko, Nick V. Pakoulin,
Alexander S. Kossatchev, and Igor B. Bourdonov

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{kuliamin,petrenko,npak,kos,igor}@ispras.ru
http://www.ispras.ru/groups/rv/rv.html

Abstract. The article presents an approach to model based testing of complex systems based on a generalization of finite state machines (FSM) and input output state machines (IOSM). The approach presented is used in the context of UniTesK specification based test development method. The results of its practical applications are also discussed. Practical experience demonstrates the applicability of the approach for model based testing of protocol implementations, distributed and concurrent systems, and real-time systems. This work stems from ISPRAS results of academic research and industrial application of formal techniques in verification and testing [1].

1 Introduction

During last decades more and more processes in industry, individual and social life fall under the influence of software. Software becomes more powerful, and the more powerful it becomes the more assured should be its safety and correctness. In this situation due to well-known human predisposition to errors various formal methods of software verification and validation acquire great importance.

Model based testing use formal models of software requirements to facilitate test development automation and is considered nowadays as one of the main instruments for software quality assurance. First approaches to model based testing appeared at the very rise of computer science. The approach based on finite state machines (FSMs) [2, 3] is one of the most widely used of them. FSMs serve as a good modelling mechanism for a long time. But modern software systems are often constructed from distributed, concurrently operating components with intention to satisfy real-time constraints. They are more complex and require more sophisticated modelling methods to capture their complexity and features adequately. Various generalizations of FSM form the foundation of many modern model based testing approaches, which are more suitable for modern software.

Often model based testing of real-time systems is based on so called timed automata [4–6], an FSM generalization augmented with temporal attributes. They are used successfully both in testing and model verification areas in research projects. But they are rarely used to model real life systems in industrial

practice. One of the possible causes of this situation is the wide gap between timed automata formalism and formalisms of programming languages used by typical developers. The same gap is inherent to the approaches based on various kinds of temporal logics [7, 8], although some commercial tools based on temporal logics are available (see [9]).

The other issue to be noticed is the usual separation of testing of event based communication from testing of unit functionality. But in practice calculations carried out in real-time systems should often be performed with timing restrictions taken into account.

We think that no model based testing approach exists that satisfies all the requirements of real-time or concurrent system testing. So, we have to try some combination of different approaches, which can be determined by three main points.

- The formalism used for description of system behaviour correctness criteria.
- Models used for test goal description and for test sequence generation.
- Means for integrated description of timed and functional (concerning calculations) characteristics of system behaviour.

From the engineering point of view one more issue is important: how to integrate the formalisms used and traditional programming techniques? The good solution of this problem implies that the resulting approach would be suitable for typical software engineers and could be smoothly introduced into industrial practice.

The approach proposed in this article is an extension of UniTesK test development method [10] successfully used for functional testing of complex industrial systems [11] on the base of formal specifications. UniTesK method is based on the following combination of techniques for solution of the problems stated above. Description of system behaviour, or behaviour model, is represented in the form of preconditions and postconditions of target system interface operations and interface data type invariants. The model used for test sequence generation is a FSM in an implicit form (see [10] for details). Usually it can be obtained by abstraction of behaviour model, but sometimes may include some more implementation specifics. The notation used for model is an extension of the programming language used in target system, and the problem of model and implementation integration does not exist.

Thus, three of four questions stated are answered by UniTesK. The open question – how to use this method for specification of timed constraints – is the subject of this article. The general idea is to extend the widely used and effective formalism of FSMs with a minimal set of features to specify concurrency and timed constraints. *Asynchronous finite state machines (AFSMs)* formally defined in the next section are considered to be appropriate for this. Before the definition we also give some background for introduction of new concepts. The third section presents testing approach based on AFSM models and corresponding modifications of UniTesK basic test architecture. Then, the conclusion summarizes the main statements of the work and outlines the future research topics. The appendix reports on some practical applications of the approach.

2 Asynchronous Finite State Machines

The classical approach for FSM based testing of software components can be illustrated by the Fig. 1. The target component is considered as a system that produces one reaction in response for one stimulus provided by the test. The task of testing is to provide a representative collection of test stimuli and to check the correctness of resulting reactions.

Fig. 1. Classical Model of Testing.

This approach works well for passive software components. It can also be adapted for active components that produce reactions only in response for stimuli, give one reaction for one stimulus, and the next reaction of which depends only on the sequence of stimuli obtained before.

But what if the target component can produce reactions without any stimuli or can give a sequence of reactions in response for one stimulus? In this case classical FSM model cannot adequately describe the target system behaviour and we have to use more sophisticated approach.

Modern software systems are usually composed from distributed and concurrently working components. Real-time software also is often designed as a collaboration of units working in parallel to satisfy real-time constraints. The testing process for such a unit can be illustrated by the picture on Fig. 2.

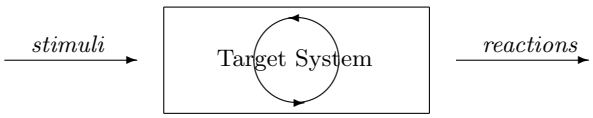

Fig. 2. Modern Model of Testing.

The difference between these two models cannot be overcome by simple interface transformation. The second model permits production of a sequence of reactions in response for one stimulus, or in absence of any stimulus. Even more complex is dependency of reaction sequence on all the history of interaction

between the system and its environment. This kind of behaviour cannot be modelled by FSM in a straightforward way.

Asynchronous finite state machine (AFSM) A is a tuple (V, v_0, I, e, O, T), where

- V is a finite set called *the set of states* of A.
- $v_0 \in V$ is called *the initial state* of A.
- I is a set called *the input alphabet* of A. Its elements are called *input stimuli* of A, or, simply, stimuli.
- $e \notin I$ is called *the empty stimulus.* The set $X = I \cup \{e\}$ is called *extended input alphabet* of A.
- O is a set called *the output alphabet* of A. Its elements are called *reactions* of A.
- $T = R \cup S \cup E$ is a finite set called *the set of transitions* of A. Its parts have the following meaning.
  - $R \subseteq V \times X \times V$ is the set of *receiving transitions.* Note, that receiving transition can be marked with stimulus from I and with the empty stimulus e.
  - $S \subseteq V \times O \times V$ is the set of *sending transitions.*
  - $E \subseteq V \times V$ is the set of *empty transitions.*

We call a state $v \in V$

- *terminal,* if it has no outgoing transitions: $v \notin \pi_1(R) \cup \pi_1(S) \cup \pi_1(E)$. We use here symbol π_i for standard projection $\pi_i : A_1 \times \cdots \times A_n \to A_i$.
- *sending,* if it has only sending or empty outgoing transitions: $v \in \pi_1(S) \cup \pi_1(E) \setminus \pi_1(R)$.
- *receiving,* if it has only receiving outgoing transitions: $v \in \pi_1(R) \setminus \pi_1(S) \cup \pi_1(E)$.
- *mixed,* if it has both receiving and sending or empty outgoing transitions: $v \in \pi_1(R) \cap (\pi_1(S) \cup \pi_1(E))$.

We consider only AFSMs without mixed states. See the definition of AFSM behaviour function at the end of this section and further discussion of possibility of mixed states.

Fig. 3 demonstrates an example of AFSM. We use here widely used notation representing stimulus a as $?a$ and reaction x as $!x$. The state 0 is its initial state of the AFSM presented. This AFSM has six states, the input alphabet $\{a, b\}$, and the output alphabet $\{x, y\}$. It also has 4 receiving transitions (one of them is marked with the empty stimulus), 4 sending transitions, and one empty transition. The state 5 is terminal.

We will show later that the behaviour of AFSM presented on Fig. 3 cannot be adequately modelled with any FSM.

To describe how an AFSM A works we need to consider *the input queue* and *the output queue* connected to it. Input queue contains a sequence of stimuli, each of which belongs to extended input alphabet X of A. Output queue collects the sequence of A reactions. Informally, A looks at the head stimulus x of its input

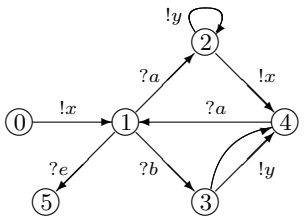

Fig. 3. Example of AFSM.

queue and tries to find a transition marked with this stimulus outgoing from its current state. If it can do this, it performs one of such transitions and removes the head stimulus from the input queue. If it cannot, possible reasons can be of four kinds: (1) the current state is terminal and A stops its work; (2) the current state is sending, so A chooses some transition marked with reaction or empty transition and performs it; (3) the current state is receiving and $x = e$, then A does not do anything, but x is removed from the input queue; (4) the current state is receiving and $x \in I$, then the error occurs and A cannot operate any longer. The empty stimulus e is introduced to model the behaviour of a system in the absence of any external stimulus, at least for some time.

A step of AFSM A in the state $v \in V$ for the first symbol x in its input queue is defined as nondeterministic choice of one of *the possible transitions,* and corresponding change of the input queue, the output queue, and the current state of A. The set of possible transitions P is determined as follows.

- If v is terminal, $P = \emptyset$. The current state of A and states of its queues remain unchanged.
- If v is sending, $P = \{t \in S \cup E | \pi_1(t) = v\}$. If the transition t chosen belongs to E, the next state of A is $\pi_2(t)$ and its queues are left unchanged. If $t \in S$, the next state is $\pi_3(t)$, the input queue is left unchanged, and the reaction $\pi_2(t)$ is added to the output queue.
- If v is receiving and $x \in X$, $P = \{t \in R | \pi_1(t) = v \wedge \pi_2(t) = x\}$. If this set is not empty, some $t \in P$ is chosen, the next state of A is $\pi_3(t)$, the output queue is left unchanged, and x is removed from the input queue. If $P = \emptyset$ and $x = e$, the state of A and its output queue are left unchanged, and x is removed form the input queue. If $P = \emptyset$ and $x \neq e$, *forbidden input error* occurs.

We consider execution of A for a sequence of stimuli put in its input queue. Such sequence can be finite or infinite. We consider only infinite sequences, because a finite sequence can be represented by the infinite one, obtained by completing the first with infinite sequence of empty stimuli. *An execution* of AFSM A for an infinite input sequence w is a sequence of steps of A, starting from its initial state v_0, while w in its input queue, and the output queue of A is empty. An execution produces some sequence of reactions u, called *the execution result,* in the output queue of A. An execution is *legal,* if it is infinite or ends in a terminal state of A. Illegal execution ends with forbidden input error.

Input sequence w is called *acceptable* for an AFSM A, if all the executions of A for w are legal. *The behaviour function* h_A of A is the function defined on the set of its acceptable input sequence. It maps such a sequence w into the set of all possible execution results of A for w.

Having behaviour functions of AFSMs we can compare them. For example, the behaviour of the AFSM on Fig. 3 cannot be modelled by any FSM, because an FSM cannot produce in response for the input sequence aa the sequence xy^kx for any given natural number k.

We can also choose a subclass $\mathcal{A}$ of all AFSMs and consider behaviour functions of AFSMs from $\mathcal{A}$. In particular, we may allow mixed states in AFSM and define its behaviour in such a state somehow. It is possible to demonstrate (see [12]) that most of the natural ways to define such generalized AFSMs result in state machine classes having the same set of behaviour functions. That is, for each 'generalized AFSM' we can construct the one satisfying the above definition and having the same behaviour function. The only exception is *input output state machines* [13, 14], which are usually defined very similar to AFSM, but has no empty stimulus. Their behaviour functions can be realized as behaviour functions of AFSMs, but there exists an AFSM that have behaviour function different from behaviour function of any finite IOSM (see [12] for an example). So, AFSMs present more general class of state machines.

It is easy to construct the AFSM having the same behaviour as a given FSM has. To do so, it is sufficient to add one intermediate state for each FSM transition and to separate each FSM transition into two ones. The first is receiving, marked with the initial transition input stimulus and leads from the starting state of the initial transition to the corresponding intermediate state, and the second is sending and leads from the intermediate state to the end state of the initial transition.

3 Conformance Testing based on AFSMs

Comparison of behaviour functions of AFSMs can be regarded as the essence of testing based on AFSM models. We say that an AFSM B *conforms* to another AFSM A if for each input sequence w acceptable for A $h_B(w) \subseteq h_A(w)$. This conformance relation is an analogue for reduction relation of FSMs [15].

Having an AFSM model of some software system we can pose a question whether the system works in accordance with the model. It can be regarded as a question on conformance of unknown AFSM, called *the implementation under test*, which we can observe only through its behaviour on sequences of stimuli, to the given one.

To answer this question with the help of testing we should impose some restrictions on the implementation under test. In absence of any restrictions for each given model and sequence of stimuli or even set of such sequences we can construct an implementation that provides the same reactions as model in response for all these sequences, but does not conform to the model. In general case we face with the following problems.

– **Nondeterminism of model.** If behaviour function maps some input sequence into several reaction sequences, then in general we cannot obtain all of them after any finite number of executions. Instead we should lay down some hypothesis on the behaviour of target system. For example, we can suppose that if it works correctly for one trial of given input sequence, it does so for each trial of this sequence. Some approaches to testing based on nondeterministic models use *all results condition* – after some bounded number of trials, we can obtain each possible output (see [16]). In an AFSM one input sequence can map to infinite set of output ones, so all results condition restricts the set of AFSM models we can use for testing.
– **Infinity of behaviour function domain.** Model behaviour function can be defined for infinite set of input sequences. We cannot test all of them, so we should define some *coverage model* – an equivalence on the behaviour function domain – and suppose that if implementation works correctly for one sequence from some equivalence class, then it does so for any one of the same class.
– **Infinity of input sequences.** We define behaviour function on infinite sequences of stimuli. How can we check implementation work for some actually infinite input (that does not end with a sequence of empty stimuli)? We do not know general solution of this problem. Some particular solution can be obtained by choice of actually finite (having only empty stimuli after some step) representatives of coverage classes. Another way is to use for check not only model behaviour function, but some additional predicates on possible serializations of stimuli and reactions, which can assign a verdict on the correctness of output on some finite step.
– **Reaction wait time.** How long should we wait for reactions? This question cannot be solved without restrictions on implementation size (number of states and/or transitions) and introduction of some bound on implementation step execution time. Under those conditions we can calculate upper bounds on reaction wait time.
– **Empty stimulus simulation.** Our empty stimulus is only an abstract representative of active nature of implementation under test. How can we simulate its presence in input sequence? We do not have general answer. One approach is to tap somehow input receipts in the implementation and to trace them. We can do so for systems, which use special function to obtain the status of input queue and for which this function can be substituted. For other systems we can use a special class of models having *time uniform behaviour.* This means that their behaviour does not differ for input sequences, which can be transformed to each other by insertion or removal of empty stimuli. Actually, this class of models is equivalent to finite IOSMs (see [12]). We can also use *the stationary testing,* which is possible when all coverage classes can be represented by sequences without empty stimuli.

We consider testing based on AFSM models in the context of UniTesK technology [10]. This technology provides a full-scale process for test development

and testing based on formal specifications of target system behaviour. UniTesK proposes to use two models in test development.

- The first one is the model of system behaviour represented in the form of pre- and postconditions of interface operations and data type invariants. Specifications are automatically transformed into *oracles* – programs that check behaviour of the system under test against specifications. First, the oracle of some target operation checks the precondition and returns precondition violation exception, if it fails. If it holds for the given arguments, the oracle calls the target operation with these arguments. Then, on the base of the result obtained, pre- and post-values of arguments and state variables, it checks the postcondition. So, such an oracle can process any values of arguments and can work with nondeterministic implementation.
- The second model is an FSM used for generation of test sequences. It is called *testing model.* Testing model is usually (but not always) an abstraction of specifications (see [17] for theoretical background of testing model construction). It is represented in the form of *test scenario,* which is an implicit description of the FSM. It has no information on ends of transitions and defines only the data structure of states and a rule of transition firing depending on state data. Test scenario can be developed manually or semi-automatically on the base of behaviour specifications and functionality coverage criterion chosen as a goal of the test.
  During the test the generic FSM traversal mechanism uses scenario to generate an adaptive sequence of input stimuli. Each stimulus is converted into its representation on behaviour model level, then with the help of adapters it is converted into implementation representation, the system produces the response on it, which should be converted back on the level of behaviour model and checked by the appropriate oracle (see [10] for details).

The peculiarity of UniTesK is the implicit representation of state machine model in test scenario. Test designer does not describe states and transitions but specify only the structure of state data (what is to be regarded as state) and input data iterators for each operation under test. The traversal mechanism that uses such a representation should be able to produce a traversal of state machine transitions without knowledge on their ends until their execution.

To use UniTesK for testing real-time and concurrent systems we use AFSM models instead of FSM for test sequence generation and extend the specification constructs. In addition to specifications of interface operations they may contain *asynchronous reaction specifications.* Such a specification also has precondition, saying in what model states this reaction is possible, and postcondition, saying what the resulting state of corresponding transition is. Postconditions can contain timing constraints.

Test suite architecture presented in [10] was also extended. To decide whether the implementation reacts correctly on the input sequence applied we do the following.

1. Collect all the reactions, maybe from several sources (implementation ports). Thus, we know some partial order on reactions, but have no full order on

them. This partial order is defined for reactions coming to one port and maybe with the help of timestamps on reactions. Note, that timestamps should be used accurately. They often contain the time of reaction arrival, not the time of its creation. So, in general reactions that arrive on different ports within some short interval cannot be surely ordered.

2. Then, we find all possible serializations of stimuli sent and reactions obtained and all possible sequences of intermediate states in the behaviour model. If we cannot find any, then implementation does not conform to specifications. If we find some, we can consider any of the resulting states as the hypothetic next state of the system and continue the process.

To do this we add to the test suite architecture the component, which generates possible serializations of partially ordered reactions and stimuli. In addition we need AFSM traversal algorithms, which can work with sets of hypothetic current states, or we may use only models that can bring us only to single state after any representative input sequence. In practice we managed to use the second approach (see below the section on applications of AFSMs). The first one is in active research.

4 Practical Applications of AFSMs

The approach to concurrent system testing based on AFSM in UniTesK framework was successfully applied to testing an implementation of IPv6 protocol. The CTesK tool based on C language extension was used in the project.

The system under test was represented by Microsoft Research implementation of Internet Protocol version 6 (IPv6), more precisely, version 1.4 of MSR IPv6 for Microsoft Windows NT/2000. The same test suite was also ported to Windows XP and Windows CE 4.1. The main features tested are the following.

- Packet send and receive, including fragmentation and reassembly of large packets.
- Control messages. As RFC 2463 [18] specifies, IPv6 used Internet Control Messages Protocol version 6 (ICMPv6) to report errors encountered in processing packets and to perform other functions, such as diagnostics. ICMPv6 must be fully implemented by every IPv6 node.
- Neighbour discovery (RFC 2461 [19]), including neighbour solicitation, neighbour advertisement, router solicitation, router advertisements, and redirect.

These features do not exhaust the functionality of IPv6. But their absence or an error in implementation of any of them in the protocol on a node often causes impossibility of normal node work in the IPv6 network.

The specifications were developed in the UniTesK-compliant specification extension of C language on the base of IPv6 specifications presented in RFC documents. The interface specified consists of a set of procedure and non-procedure stimuli and two types of reactions: outgoing IPv6 packets and UDP messages

received in sockets. One additional reaction was introduced to specify timing restrictions, namely, timeouts in neighbour discovery protocol.

The following defects were discovered.

- Some series of packets can cause reboot of the system. The error in packet reassembly implementation causes in some cases fatal error in the operating system core.
- A discrepancy between specifications of packet reassembly from fragments and its implementation. This defect was classified as negligible, because it occurs only under specific conditions and does not cause fatal consequences.
- A discrepancy between specification of ICMPv6 Echo function and its implementation. RFC requires that all the responses on ICMPv6 Echo request should be passed to the process initiated the request. MSR IPv6 implementation interface supports passing only of the first response.

The project demonstrated the capabilities of UniTesK approach extended with AFSM models for specification and testing concurrency and real time constraints. The testing performed revealed several defects, while several projects on MSR IPv6 testing carried out simultaneously do not report on their discovery (see [20] for example). The project effort was about 10 man-months, which is much less than the effort of implementation.

The project also revealed some disadvantages C language for representation of abstract software properties specifications. The first is very low level of C language, which hides the logic of abstract model, the second one is manual memory management, which requires much work not related with the matter of specification and test design.

For details see full project reports on [21].

The second project performed according to the method presented in this article is testing of OPC RACE, a peer-to-peer messaging management systems for Nortel Networks. During this project also several errors were found in the implementation.

5 Conclusion

The UniTesK approach for model based testing tries to use notation and concepts familiar for typical developer. This principle was accepted as a compromise that facilitate introduction of formal methods into industrial processes. Sometimes it requires simplification or hiding of theoretical background and advanced techniques used, sometimes forces to exclude promising sophisticated test development techniques from consideration. It provides software engineers with a model based testing technology of moderate complexity that do not require special modelling languages. The article demonstrates the next step in the same direction. The techniques for modelling of concurrency and timing characteristics are represented in an environment familiar for industrial developers.

Introduction of AFSMs in the background of UniTesK technology does not seem to make it much more complex, while gives a set of features especially useful for testing of real-time and concurrent systems. On the base of experience of

practical applications [21–23], we can say that the approach is quite effective for specification and further conformance testing of protocol implementations, distributed and parallel systems, for testing both functionality and time constraints of real-time systems.

The UniTesK technology is supported by a set of tools developed for different target languages. We already include the support for AFSM based testing in CTesK tool, working with an extension of C, and plan to do so for J@T and .N@T tools, working with Java and C# extensions correspondingly [24, 25].

References

1. http://www.ispras.ru/~RedVerst/
2. George H. Mealy. A method for synthesizing sequential circuits. Bell System Technical Journal, 34(5):1045-1079, 1955.
3. E. F. Moore. Gedanken-experiments on sequential machines. Automata Studies, Annals of Maths. Studies, Princeton University Press, no. 34, pp. 129-153, 1956.
4. A. En-Nouaary, H. Fouchal, A. Elqortobi, R. Dssouli, and E. Petitjean. Timed Testing Using Clock Zone Vertices. Technical Report, Departement d'IRO, Universite de Montreal, 1998.
5. D. Clarke and I. Lee. Automatic Test Generation for the Analysis of a Real-time System: Case Study. In 3-rd IEEE Real-time Technology and Applications Symposium, 1997.
6. B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. International Journal on Software Tools for Technology Transfer (STTT), 2002.
7. F. Dietrich, X. Logean, and J. P. Hubaux. Testing Temporal Logic Properties in Distributed Systems. Proc. IFIP Int'l. Wksp. Testing of Commun. Sys., Tomsk, Russia, Aug. 1998.
8. H. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002), April 8-11, 2002.
9. http://www.time-rover.com/main.html
10. I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.
11. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. FM'99: Formal Methods. LNCS 1708, Springer-Verlag, 1999, pp. 608–621.
12. I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. Classification of Asynchronous Finite State Machines. In Russian. To be printed in works of ISP RAS.
13. P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand. Towards Analysing and Synthesizing Protocols. IEEE Transactions on Communications, COM-28(4):651-660, April 1980.
14. O. Henniger. On test case generation from asynchronously communicating state machines. In M. Kim, S. Kang, K. Hong, eds., Proceedings of the 10-th IFIP International Workshop on Testing of Communicating Systems, Cheju Island, South Korea, 1997. Chapman & Hall.
15. G. von Bochmann, A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. Proceeding of ISSTA 1994, pp. 109-124.

16. S. Fujiwara and G. von Bochmann. Testing Nondeterministic Finite State Machine with Fault Coverage. IFIP Transactions, Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems, 1991, Ed. by Jan Kroon, Rudolf J. Heijink, and Ed Brinksma, 1992, North-Holland, pp. 267-280.
17. I. Burdonov, A. Kossatchev, and V. Kulyamin. Application of finite automatons for program testing. Programming and Computer Software, 26(2):61–73, 2000.
18. A. Conta and S. Deering, *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification,* RFC 2463, December 1998.
19. T. Narten, E. Nordmark, and W. Simpson, *Neighbor Discovery for IP Version 6 (IPv6),* RFC 2461, December 1998.
20. http://www.tahi.org
21. http://www.ispras.ru/~RedVerst/RedVerst/White Papers/MSRIPv6 Verification Project/Main.html
22. A. Petrenko, I. Bourdonov, A. Kossatchev, V. Kuliamin. Experiences in using testing tools and technology in real-life applications. Proceedings of SETT'01, India, Pune, 2001.
23. A. K. Petrenko. Specification Based Testing: Towards Practice. Proc. of PSI 2001. LNCS 2244, Springer-Verlag 2001.
24. http://www.atssoft.com
25. http://unitesk.ispras.ru