ICIT 2015 The 7th International Conference on Information Technology
doi:10.15849/icit.2015.0096 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

Symbolic Modeling Approach in Verification and Testing

Oleksandr Letychevskyi
Department of Theory of Digital Automatic Machines
Glushkov Institute of Cybernetics of National Academy of Sciences
Kyiv, Ukraine
lit@iss.org.ua

Abstract—The paper outlines a symbolic modeling approach developed in Glushkov Institute of Cybernetics and applied in verification and model-based testing. This method is the result of 10 years of experience in a large amount of industrial projects in different subject domains. The models in this approach are presented as UCM (Use Case Maps) notation composed with basic protocols formal language. Symbolic modelling is used in verification of requirements and models of programs. It is also intended for creation of test suits and further test execution.

Keywords—symbolic modeling, symbolic execution, model-based testing, verification of requirements, predicate transformers

I. INTRODUCTION

The paper describes a symbolic modeling approach developed in Glushkov Institute of Cybernetics. Usage of symbolic modeling is the result of over 10 years of experience in the industrial application of formal methods in software development process especially for the requirements gathering and testing stages. It was deployed in a large number of projects at Motorola and Uniquesoft LLC in various subject domains, from telecommunications to networking, microprocessor programming, and automotive systems. The structure of the paper is the following.

In section 2 we consider related works dedicated to usage and development of deductive technologies and symbolic approach in software industry especially verification and testing. Section 3 describes the formal specifications implemented in UCM (Use Case Maps) notation and basic protocols language that are the input of our symbolic modeling method. Section 4 outlines the theoretical basics and semantics of symbolic modeling methods especially theory of predicate transformers developed in Glushkov Institute of cybernetics. Sections 5 - 7 describe usage of symbolic modeling in verification of requirements, test generation and test execution.

II. SYMBOLIC METHODS IN VERIFICATION AND TESTING

Usage of symbolic-based formal methods has become relevant due to the growth in complexity of the designs common in hardware and software industries. The term “symbolic” is closely related in meaning to “algebraic” where manipulating with mathematical expressions is anticipated. The methods of symbolic execution or symbolic modelling were invented in 70-th [1] but they became popular only now due to essential success in solving and proving techniques development.

SMT (Satisfiability Modulo Theory) solving approach was invented as a trend of model checking technique. Model checking [2] is an exhaustive exploration of the states and transitions of the mathematical model. Term “symbolic model checking” has been introduced by McMillan [3] where states of model are presented as formulas in some theory. It subsumes different SMT techniques and similar methods such as abstract interpretation [4], symbolic simulation [5], abstraction refinement [6], and others. Model checking is supported by a number of tools, such as SPIN or BLAST. The main objective of SMT-solving usage is a handling of states explosion problem while traversing the model states.

Usually SMT approach is applied in verification as a formal proof of properties of an abstract mathematical model of the system captured by the requirements or a program labeled by annotations. Examples of such mathematical models are finite state machines, labeled transition systems, Petri nets, or process algebras. There are two main approaches to establish properties of such models. In difference with exhaustive searching of the states set the other approach is deductive verification where the model or program specifications are presented as a set of assertions and the properties are established by theorem provers such as HOL, Z3, CVC or Isabelle.

Anyway, the detection of property true is defined by reachability of states that present this property. So the proving-based methods are complemented by the modelling features. The problem of reachability is unresolvable for systems with infinite number of states and methods of its detection were invented as heuristics for different classes of models.

So far the only few tools were presented as symbolic modeling for model-based testing. RT-tester [7] is commercial tool for symbolic trace generation with usage of SMT-solvers. Symbolic execution is also used for test generation in Symbolic Path Finder [8] for Java applications.

The main purpose of model-based testing is to obtain efficient coverage of model behavior. It is anticipated that the set of generated tests should cover the maximal set of states of models. There is a big variety of commercial MBT tools but they are not coping with huge or infinite number of states.

The more significant achievements in symbolic execution usage are the SAGA-project [9], KLOVER [10]. It implements symbolic modeling for detection of defects in code executing it symbolically.

The use of symbolic and deductive methods is difficult for engineers due to the complexity and unfamiliarity of formalization and verification methods. So one of the challenges aim at making symbolic techniques more amenable to industrial usage by front-ending mathematical techniques with familiar or easy to learn notations, including expressive graphical presentations.

III. PROBLEM STATMENT AND INPUT MODELS

We consider two problems for application of symbolic modeling methods. The first is verification of models that present the program code or design models or requirements specifications.

Verification procedure is a searching of given property (or property violation) while model behavior. For example, it could be requirements specifications where the properties violations are inconsistency, incompleteness, safety violation or liveness. In a program code, it could be bounds violations, null pointer assignment or other possible errors.

The second is model-based testing where we have the model for test generation and problem is to generate the number of tests due to the customer request. Such request could contain coverage criterion that defines quality of test procedure as necessary covered states of model.

In both cases, we face with procedure of model formalization. It is the most time-consuming process because of manual creation of formal specifications from natural text or semi-formal descriptions, or other source of initial model presentation.

There is a number of widespread modeling notations such as UML2, SysML, ASM, Timed automaton (TASM). We consider the model of system as a composition of two notations. The base is UCM (Use Case Maps) notation standardized as part of URN (User Requirements Notation) in ITU-T recommendation (Z.151) [11] that provides a scenario-based approach to requirements specifications. UCM allows an easy and natural expression of sequences of events with synchronization and structure. The language of basic protocols developed at the Glushkov Institute of Cybernetics [12] extends UCM. It represents behavioral scenarios as reactions of a system to externally triggered events under some conditions. Such local behaviors are modeled by basic protocols that consist of three components:

- precondition defines the state of the environment of the system at the point when the basic protocol is applicable;
- process actions are presented as MSC (Message Sequence Chart) diagrams that show input and output signals and local actions;
- postcondition defines the change of the environment in response to the application of the basic protocol.

Pre- and postcondition are represented as formulae of the basic logic language. It supports attributes of numeric and symbolic types, arrays, lists, and functional data types. The following example of a basic protocol (Fig.1) is taken from the specification of a well-known telecommunication protocol.

The order and synchronization of basic protocols is defined by means of UCM diagrams in a graphical notation. UCM diagrams are oriented graphs with initial and final states. Nodes of the graph represent events in a system.

The basic protocols notation captures the atomic actions (responsibilities) of a UCM map. The UCM notation is a convenient tool for the description of parallel processes and

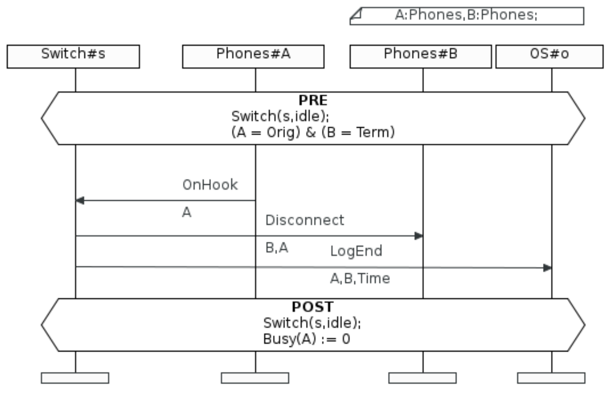

Fig. 1. Example of basic protocol as a part of Plain Old Telephone System

their synchronization. An example of a UCM diagram is given below (Fig 2.).

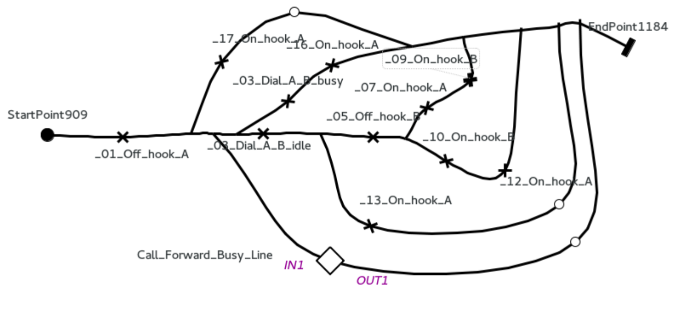

Fig. 2. Use Case Maps for part of telephony protocol

Models that created as composition of UCM present the models of formal requirements that are the source of test generation. This method could be extended for model of program where UCM defines a control flow of program and basic protocols contains pre- and postconditions for program statements. Such model could be verified with applying of methods of symbolic modelling.

IV. SYMBOLIC MODELING SEMANTICS

The model of a system is considered as a hierarchical set of interacting environments and agents *inserted* (exist) into these environments [13]. The model description has three main levels: basic protocols level, UCM level, and modelling level. Each level can be considered as a level of abstraction in system description. The most abstract is a basic protocols level. It contains the largest behavior describing the evolution of a system. The control level restricts the behavior or set of scenarios strengthening the environment control of agent behaviors according to the general requirements to the system. The modelling level or level of trace generation is the least abstract and depends on a problem solved on a base of the model of a system. It provides further restriction of the set of possible traces generating.

Basic protocol level constitutes with the set of basic protocols. Each basic protocol describes one of the possible elementary scenarios of system behavior. Algebraic form of basic protocol is a formula

$$\forall x\ (\alpha(x) \rightarrow <P(x)>\ \beta(x)) \qquad (1)$$

In this formula $x=(x_1,x_2,\ldots)$ is a list of typed parameters, $\alpha(x)$ and $\beta(x)$ are the formulas of logic language called the *basic language* of a model, $P(x)$ is a finite process, which describes the interaction of agents and their environment by message passing. Formula $\alpha(x)$ is called a precondition, and formula $\beta(x)$ is called a postcondition of basic protocol. Basic protocol can be considered as a temporal logic formula that expresses the fact that if (for suitable values of parameters) a system state satisfies the condition $\alpha(x)$, the process $P(x)$ can be activated and, after its successful termination, the new state of a system will satisfy the condition $\beta(x)$. We use MSC language [14] for graphical representation of basic protocols.

Basic language of a model is first order multityped (multisorted) logical language. Simple types are numeric (real and integer), Boolean, symbolic, and enumerated types. There are two kinds of functional symbols in basic language. The first kind consists of the symbols with fixed interpretation (arithmetic operations for numeric, logical connectives for Booleans etc.).

The second kind of functional symbols change their interpretation during the evolution of a system like Abstract State Machine [15]. These functional symbols are called attributes. Attributes of arity zero have simple types and are called simple attributes. The attributes of arity more than zero are called functional attributes. They have fixed types for domain and range values and are used for the representation of data structures like arrays. The access to the values of these data structures provided by the attribute expressions like $f(t_1,t_2,\ldots)$ where f is an attribute symbol and $t_1,t_2,\ldots$ are attribute or constant expressions. If all arguments of attribute expression are constant expressions then attribute expression is called a constant attribute expression.

In the set of functional symbols of basic language there are also some distinguished collections of attributes which are called agent attributes. Each collection of agent attributes defines agent type. Agent types define the types of agents, which can be inserted into given environment. The description of environment attributes and the types of agents constitutes environment description and defines the environment type. A pair $<E,L>$ consisting of environment description E and the set L of basic protocols consistent with this environment description constitutes the description of a system on basic protocol level.

Let $<E,L>$ be the description of a system. For this description we define a transition system $S=S(E,L)$, which describes the evolution and all possible traces of this system.

The state $s=\sigma[u_1,u_2,\ldots]$ of a system S is represented as a composition of a state of environment σ and the states $u_1,u_2,\ldots$ of agents inserted into this environment. The state of environment can be concrete or symbolic. Concrete state is the mapping from the set of constant attribute expressions (including agent attribute expressions) to the set of their values (consistent with the types of attribute expressions). Symbolic state is a formula of basic language.

The state of agent is defined by the values of constant attribute expressions of agent attributes in concrete state, and the properties of agent attributes in the formula of symbolic state.

Each of basic protocol B defines some transition relation $s \xrightarrow{B} s'$ on the set of concrete system states. To compute this relation one must first instantiate the basic protocol by substituting some concrete values of parameters to basic protocol. If the precondition of basic protocol is true then basic protocol B is applicable to the state s and the state s' is selected non-deterministically among those states on which the postcondition is true. Transition system defined in this way is called a concrete model of local description level. To make a concrete model more deterministic the assignments and conditional assignments are allowed in postcondition. To obtain a trace of a concrete model starting from some initial state s_0 one of the possible histories of a system evolution

$$s_0 \xrightarrow{B_1(z_1)} s_1 \xrightarrow{B_2(z_2)} \ldots \xrightarrow{B_n(z_n)} s_n \qquad (2)$$

for basic protocols instantiated by $z_1,z_2,\ldots$ is used to create a trace $P_1(z_1)*P_2(z_2)*\ldots*P_n(z_n)$ as a weak sequential product of instantiated MSC charts for basic protocols processes.

Symbolic model of basic protocols level is a transition system with formulas of basic language as states. Transition $s \xrightarrow{B} s'$ for basic protocol $B=\forall x\ (\alpha(x) \rightarrow <P(x)>\ \beta(x))$ is computed as follows. First check the satisfiability of the formula $s \wedge \alpha(x)$ where the elements of a list x are considered as the new simple attributes of environment description. If the formula is satisfiable, then compute $pt(s \wedge \alpha(x),\beta(x))$. The function pt is called predicate transformer and it computes the strongest postcondition provided that precondition before computation was $s \wedge \alpha(x)$. The details for computing the predicate transformer for formulas with quantifiers can be found in [16]. Generating traces from histories is made in a similar way as for concrete model but instantiation is needed only for the names of agents, which are used to name the instances in MSC diagrams.

It is not sufficient to use only basic protocols level for complete specification of a model. The matter is that constraints on sequences of application of basic protocols are not defined on them, which can lead to the consideration of undesirable histories and traces. The next level of a model description consists of the definition of a succession relation on the set of basic protocols. This relation can be introduced by the definition of additional control attributes and conditions limiting the conditions of application of basic protocols on these attributes. An inconvenience of this description is the need for the partition of the basic attributes and auxiliary control attributes. Moreover, the basic protocols themselves become more complicated. Therefore, it is useful to construct the control level as the separate upper level system.

Semantically control system over local description level can be defined as a transition system with the set of actions that includes besides of the own actions the references to basic protocols of a low level system. The control system is considered as an environment for low level system and its state is described by expression *U[s]* where *U* is a state of a control system and *s* is a state of a low level system *S=S(E,L)*. The transition rules for control system over *S* can be described now by the following rules:

$$\frac{U \xrightarrow{a} U'}{U[s] \xrightarrow{a} U'[s]} a \notin L \tag{3}$$

$$\frac{U \xrightarrow{a} U', s \xrightarrow{a} s'}{U[s] \xrightarrow{a} U'[s']} a \in L \tag{4}$$

If we replace the first rule by the rule

$$\frac{U \xrightarrow{a} U'}{U[s] \rightarrow U'[s]} a \notin L \tag{5}$$

with hidden transition then an external observer will not find any difference between the functioning of the low level system with control and without control. He will see only traces, and they are traces of the same low level system. However, with the use of the control system, their number can be smaller, and also deadlock states can arise that are absent in the case of the low level system if the control system is not correct.

V. Symbolic verification

We consider the verification of requirements for reactive systems presented as combination of UCM notation and basic protocols.

It supports the following kinds of verification. *Checking the consistency* of requirements means detection of non-determinism and ambiguities in behavioral requirements. Often such issues are deeply hidden in specifications and could entail subsequent errors and misunderstanding by developers. *Incompleteness detection* helps finding of deadlock situations in formal model of requirements.

Different from concrete modeling, symbolic methods involve deductive systems such as provers or solvers. Formal model could present the system at a high level of abstraction where deductive techniques are most suitable. Deductive systems provide proofs of assertions in a first-order theory, resultant from the integration of theories of integer and real linear inequalities, enumerated data types, uninterpreted function symbols, and queues. This technique allows the verification of the model for systems with a large or infinite number of states.

Symbolic techniques also support incremental verification that is important for the development of large systems. Different parts of formal specifications can be verified separately and encapsulated into enlarged entities to avoid repeating the verification of such components. For a system with high degree of features interactions each feature can be verified separately first and their interactions can be verified without examining the individual behaviors repeatedly.

Author is one of the developers of IMS (Insertion Modeling System) developed in Glushkov Institute of cybernetics. Symbolic simulation in IMS allows detecting reachability of the violation of correctness properties by considering symbolic states of a system and generating a set of traces leading to the findings.

A static requirements checking in IMS is intended for detection of candidates for properties violations. It is based on formal proving of statements and involves deductive tools. If property violation is detected then its reachability shall be proved by means of symbolic modelling.

The system establishes a trace that leads to a deadlock or other anomaly situation and identifies its causes. A trace is graphically presented as a MSC diagram. Safety violations in IMS also are detected by means of symbolic modeling or static proving. Liveness of a system is checked by finding the reachability of the necessary property. Livelock detection identifies situations where a system may or otherwise be non-responsive.

VI. Symbolic test generation

The formal presentation of requirements in UCM notation together with basic protocols gives the possibility to generate a test suite at the given level of abstraction in IMS system. A set of traces can be generated from formal requirements that will be obtained in MSC notation and can be converted into standard test formats.

Traces contain input and output signals and local actions of the system together with the set of states of the environment that contains possible values for the system attributes. These states are symbolic and cover potentially large sets of attribute values. The generation of traces corresponds to different coverage types defined in a *user trace generation request.*

Node coverage. Coverage of all nodes constructs in UCM diagram that corresponds to coverage of all functionalities of model or coverage of requirements.

Edge coverage. Coverage of all adjacent pairs of UCM responsibility nodes or other transition points.

Path coverage. Coverage of all paths those are possible between start and end points with restriction on the length of a path or the number of visiting the same nodes of UCM. Path is a sequence of "responsibility" constructs presented by basic protocols.

Full state coverage. Coverage of all states of the system that can be reached by the restricted number of steps.

The request for trace generation could be extended by selection of starting or end points or excluding of UCM nodes or edges. The traversal of constructs "stubs" that present references to other UCM diagrams also could be tuned by definition of mentioned coverage type for itself. The strategy also could be defined by length of generated traces. It could be generated as the shortest distance between start and end points or cover the maximal number of applied basic protocols.

Unreachable edges of UCM diagrams could occur during trace generation. It could happen due to the insufficient adjustments of trace generation for instance insufficient number of loops or maximal length of trace.

After trace generation, there are number of unreachable nodes and edges of UCM diagrams. Their unreachability should be proved or refuted by the following possibilities:

Backward trace generation. We generate backward trace by means of backward predicate transformer from candidates for unreachability to known reachable points. If backward generation will finished at finite number of steps by traversal of all possible paths and will not reach given point, then this point is unreachable.

Invariants usage. During trace generation the set of invariants could be generated for every basic protocol. We consider preinvariant – formula that defines possible states of environment before basic protocol application, and corresponding postinvariant after. We consider invariant as overapproximation of set of states or the weakest invariant formula. If intersection of invariant formula and precondition of successor is empty then the successor is unreachable.

All generated traces forms the test suite for further usage in test execution. Every trace is MSC diagram marked by the formulas of environment state.

Test could be built by selection of the instance for SUT (system under testing) and instance for testing system in MSC diagram. Message in MSC diagram defines the points of interaction between SUT and testing system. Formula defines the set of possible parameters of output message from testing system and oracle set of values for comparison with input messages as reaction of SUT.

For testing procedure the concretization of attributes shall be provided. It is performed by solving of constraints presented by given environment formulas. Then testing procedure or test execution is implemented by launching of tested program under control of input values from generated tests. Test suite could be converted to standard test format for instance TTCN format and be executed on existing standard test execution tools.

VII. Symbolic test execution

Method of concretization of symbolic traces is useful when test model and tested program are on the same level of abstraction. If test model is given on a higher level of abstraction then concrete values could not be defined correctly. The other disadvantage is that some scenario of behavior could be missed due to the high degree of detailing of program code.

To overcome these disadvantages the symbolic test execution is proposed. It based on the symbolic execution of tested program correspondingly to generated test.

The scheme of symbolic testing is the following.

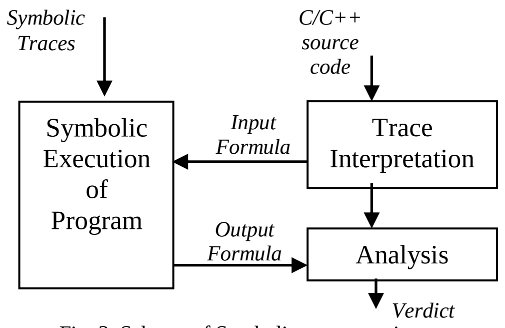

Fig. 3. Scheme of Symbolic test execution

The input of symbolic testing procedure is the set of generated symbolic traces as MSC marked by formulas over parameters of messages, and source code of system to be tested. The program of symbolic execution starts SUT performing correspondingly to scenario given in MSC trace. It changes the code environment under control from tested system that is also defined by formula over program variables. SUT accepts input data as formulas and symbolically executes statements. It returns the output as formula of code environment state and the following analysis is performed.

Let us consider some cases of symbolic test execution. Let the oracle of the code environment is a formula X and symbolic state of code environment is Y.

- If the conjunction of Y and $\neg X$ is not satisfiable then test is passed.
- If the conjunction of Y and $\neg X$ is satisfiable then there exist an unpredictable concrete state not covered by X and the formula $Z=Y\vee\neg X$ covers all such states.

In the last case, there are two possibilities.

- The conjunction of X and Y is unsatisfiable (equal to 0). All possible concrete states are unpredictable, the testing process failed.

- The conjunction of *X* and *Y* is satisfiable. The testing process can be continue, however system signalizes about the existence of unpredictable states.

Backward symbolic moving via given test from the current set of unpredictable states of environment covered by the formula *Z* allows detecting the error cause.

Symbolic test execution is based on symbolic execution of program under control of test. We consider C/C++ source code and encode it to the UCM notation with basic protocols. The corresponding interpretation in the term of basic protocols is implemented for C/C++ control statements, assignment and condition statements with linear arithmetic, calls of functions, and all kinds of cycle statements. Address arithmetic and pointers, structures, arrays, unions with integer, long, real, character, double data types also are realized. The pre- and postconditions were created for functions from input/output, memory, standard libraries.

We consider two next kinds of symbolic test execution. The first is symbolic "black box" online testing where we use the test model for generation of behavior of SUT and compare its generated oracles with symbolic environment of the executed program code. In this case we are trying to generate coverage for test model disregarding code coverage.

The scheme of "black box" online testing is the following.

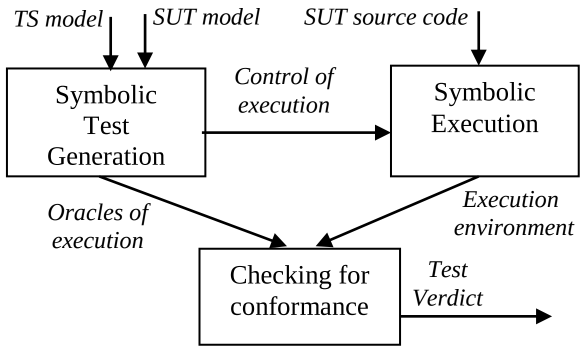

Fig. 4. "Black box" online symbolic testing

Here we generate the traces for parallel composition of TS (tested system) and SUT (system under testing). The generation is controlled by symbolic execution of SUT source code correspondingly to generated traces. The obtained states of environment are compared for conformance in checking module.

There are a number of requirements for implementation of such technique. One of this is that the level of abstraction for interchanged signals for source code shall be the same as for test (requirements) model. It means that the set of signals shall be the same for both models. The correspondence between names of variables of code and names of system attributes also shall be defined.

The other kind of online testing is symbolic "white box" testing where we execute source code symbolically checking the conformance of constraints from generation in requirements model and symbolic environment of executed program source code. In this case we are trying to cover all branches of source code. The scheme of "white box" testing is the following:

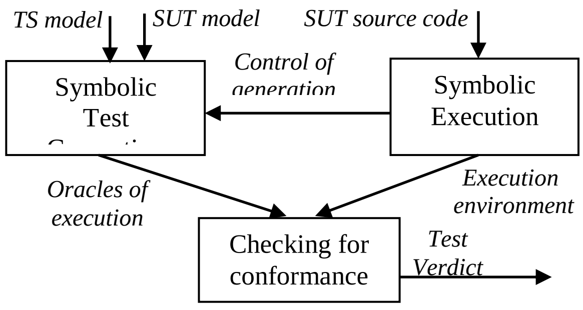

Fig. 5. "White box" symbolic testing

The difference is that source code symbolic execution controls the trace generation and compare for conformance the corresponding constraints.

Note that for requirements models of high level of abstraction the symbolic test execution is insufficient. In this case test execution with concrete values could be useful additionally.

REFERENCES

[1] J. C. King, "A new approach to program testing," in Proc. Int. Conf. Reliable , Software, Apr. 1975, pp. 228-233.

[2] Doron Peled, Patrizio Pellicone, Paola Spoletini, "Model Checking", Wiley Encyclopedia of Computer Science and Engineering, 2009.

[3] Kenneth L. McMillan, "Symbolic Model Checking", Kluwer Academic Publisher, 1993.

[4] Patrick Cousot, "Formal Verification by Abstract Interpretation", Lecture Notes in Computer Science, 2012, vol. 7211, pp. 3-7, Springer.

[5] Robert B. Jones, "Symbolic Simulation Methods for Industrial Formal Verification", 2002, Springer.

[6] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, "Counterexample-Guided Abstraction Refinement", Lecture Notes in Computer science Volume, 1855, 2000, pp. 154-169.

[7] J.Peleshka, E.Vorobev, F.Lapschies, C.Zahlten, "Automated Model-Based Testing with RT-Tester", Technical report, 25.05.2011.

[8] C. Pasareanu, N.Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode", Proceeding ASE '10, pp.179-180.

[9] Patrice Godefroid, Michael Y. Levin, David Molnar, "SAGE: Whitebox Fuzzing for Security Testing", Magazine Queue-Networks, Vol. 10, Issue 1, 2012.

[10] Guodong Li, Indradeep Ghosh, Sreeranga P. Rajan KLOVER: "A Symbolic Execution and Automatic Test Generation Tool for C++ Programs", Lecture Notes in Computer Science, Volume 6806, 2011, pp.609-615.

[11] ITU-T Recommendation Z.151, "User Requirements Notation (URN) – Language definition", 10.2012.

[12] A.A. Letichevsky, J.V. Kapitonova, V.A. Volkov, A.A. Letichevsky jr., S.N. Baranov, V.P. Kotlyarov, T. Weigert "System Specification with Basic Protocols", Cybernetics and System Analyses. 2005, № 4, pp. 3–21.

[13] Letichevsky A., Gilbert D., “A Model for Interaction of Agents and Environments”, Lexture Notes in Computer Science, 1999, №182, pp. 311-328.

[14] ITU-T Recommendation, Z.120, Message Sequence Charts.

[15] E. Borger, R. Stark, ”Abstract State Machines”, Springer-Verlag, 2003.

[16] A. Letichevsky, A. Godlevsky, O. Letychevskyi, S. Potienko, V.Peschanenko, “The properties of predicate transformer of the VRS system”, Cybernetics and System Analyses, №4, 2010, pp.3-16.