Multi-paradigm Models as Source for Automated Test Construction

Victor V. Kuliamin [1,2]

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
http://www.ispras.ru/groups/rv/rv.html

Abstract

The article discusses problems of model based test construction and ways of their solution using different kinds of models (operational, contract, axiomatic, and history-based specifications). The main idea is that the integration of model based techniques having different underlying formalisms can give valuable practical results in test construction. The idea is illustrated by successful applications of UniTesK test development technology based on the combination of contract specifications used to describe system behavior and operational models used for test sequence generation. UniTesK was designed in RedVerst [1] group of ISP RAS on the base of experience obtained in several industrial software testing projects.

Key words: Model based testing, specification based testing

1 Introduction

More and more of critical human activities in the contemporary world are falling under control of software. Such situation requires effective methods to ensure high level of software reliability and other aspects of its quality.

Testing is one of the most useful methods of quality checking. But with growth of the software complexity the effort needed to test it thoroughly seems to grow according to some nonlinear law.

Software engineering has developed a variety of techniques based on abstraction and various kinds of composition to construct complex software systems. They help to make the growth of effort needed for software construction more manageable. Unfortunately, now we lack similar techniques for software

[1] This work is partially supported by RFBR grant 02-01-00959, by grant of Russian Science Support Foundation, and by Program 4 of Mathematics Branch of RAS.

[2] Email:kuliamin@ispras.ru

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

verification and validation. So, we are able to verify or test thoroughly only systems smaller and simpler than the ones we can construct.

Model based testing is one of the promising approaches to checking quality of extremely complex systems. Its main idea is to use simple models abstracting the target system to construct tests for it. Model based testing approach allows high degree of test development automation because the models used can be represented in a formal way and can be processed by computer. On the other hand models themselves may represent different aspects of the system under test, for example, requirements and design decisions, making possible their collation to measure various quality factors of the target system.

But theoretic advantages of model based testing do not lead straightforward to its usefulness in practical applications. It is not quite obvious that model based methods can be applied successfully to complex software systems produced in the industry. Moreover, a single success in such an application does not mean that the techniques used can be easily extended to large variety of industrial software. All these problems can be solved only by experimentation and finding out the characteristics of the approach critical for its success in industrial practice.

RedVerst [1] team of ISP RAS proposes UniTesK test development technology as an example of model based testing method suitable for use in the industry. Our confidence in its suitability is based on 10-year experience of application of similar techniques in the industrial software verification projects. But it also has a base in the unique features of the technology itself.

Industrial use of model based testing methods needs the full-scale support of all aspects of test development and testing. UniTesK provides such a support as it is shown in the main part of the article. Moreover, it seems that to obtain valuable practical results of formal models we need to combine different formal methods. UniTesK is an example of multi-paradigm technology where different approaches to formal treatment of software live in symbiosis and only being used together they are able achieve the goals stated.

Further sections of the article consider various problems related with test development and testing on all the phases of software lifecycle. The next section provides some description of problems to be solved by the test development technology claiming its applicability to industrial software. The section also briefly depicts existing solutions of these problems in various formal frameworks. Its main conclusion is that no single formal approach gives all the features we want to have in the successful test development technology. The third section presents the main ideas of UniTesK approach and shows how different formal techniques are integrated in it. The last section provides some conclusions and describes possible directions of future work.

2 Problems of Model Based Test Development

Main problems of test development.

Each testing technology claiming its applicability to the wide variety of industrial software has to deal with the following problems.

- *Problem of correctness checking.*
  When we want to test something, we usually want to check if it works 'properly'. For simple and small systems we usually know very well what the 'proper' behavior is and what is not. For complex systems it is not only difficult to check that their behavior is correct, but also the correct behavior itself usually requires to be defined more precisely, because it is too hard to have in mind all the possible variants of system's behavior at once. Testing should somehow take this precise definition into account.
- *Problem of test quality assessment.*
  The full-scale method of test development should include means to measure the quality of resulting tests. Such means are usually represented as test coverage criteria. The method based on models of the target software should contain definitions of test coverage criteria based also on models, maybe the same. Presence of test coverage criteria allows so called *coverage-driven testing,* that is construction of tests in such a way that maximizes the coverage obtained.
- *Problem of test sequence construction.*
  For systems, which produce output only on the base of their input, high quality tests can be constructed only using separate test actions. This is not the case for software systems, which behavior depends on the history of their interaction with environment, i.e. on previously applied inputs. For such a system each good test coverage criterion considers system states, which accumulate all actually necessary information on the history. To test them thoroughly we need to construct sequences of test actions. The task becomes even more complex when the set of permitted actions depends on the results of the previous operations. Such situations are rather ordinary for industrial software and should be embraced by test construction methods intended to be used there.
- *Problem of changes.*
  It is well known that software and industrial software in particular is not static entity, it is always changing. Testing technology cannot be considered as a serious candidate for industrial use if it does not deal with possible changes in the target software and in the environment and does not contain special techniques to cope with them. The common method used to minimize influence of changes in software construction is *abstraction.* We will see that the same approach is widely used and provides good results in testing too.

- *Problems of testing concurrency.*
  The useful testing technology should support widely used features of contemporary software. If we consider the most 'popular' kinds of software systems, which are actively developed in the modern industry and supported by modern development technologies, we find concurrent and distributed systems. For sequential systems interpretation of their reactions in response to external action is rather simple and straightforward. Concurrency (which is also the main characteristic of distributed systems) being considered from the viewpoint of model based testing gives rise to a problems even in this seemingly simple domain. More hard become the previously mentioned problems of test development when we try to solve them for concurrent systems.

  There are a lot of formal methods that can be applied to description, construction, or analysis of concurrent systems. But authors do not know references on some formal and at the same time full-scale consideration of testing issues for concurrent and distributed systems. First, if we have a model of such a system, how to stimulate test actions for it in a way that explicitly check its concurrent behavior? And how to interpret observable system responses in order to check their conformance to the model given? Here we face with unclear notion of concurrent actions and impossibility to introduce a full order on the input actions and system responses, if they are observed in different places, which is ordinary for concurrent and distributed systems.

  Most test development techniques targeted for concurrent and distributed systems are based on models that hide system concurrency and describe its operation in terms of mostly sequential actions and responses. The only concurrency-related feature in such models is their nondeterminism. We think that this is far from the appropriate solution and concurrent actions should be explicitly addressed when we are trying to develop good tests for a concurrent system.

  More subtle is the question on what kinds of models give the most clear and practically useful answers on the previous questions from the testing viewpoint. Of course, it requires a lot of case studies and comparison of applications of various methods. But now it is not even considered by the researchers.

Below in this section we briefly consider existing solutions for the problems stated.

2.1 Behavior Correctness Checking

The well known solution for the problem of automatic correction checking is *test oracles.* Test oracle in general is a program, which evaluates the work of the target system and decides if it works correctly or not. The good source to construct oracles is formal specifications stating the requirements to program behavior in the formal way. Such specifications are usually more clear and reusable than the oracles themselves and can be processed by computer, so the oracle development can be automated.

Different kinds of specifications give rise to different methods of oracle construction. Below we regard the main such methods in short. The reader interested in more exhaustive review and technical details may refer to [2].

- *Operational specifications.* Operational specifications describe the system behavior in the form of explicit sequences of operations to be performed. Operations used should be simple enough to have obvious and precise semantics. At the same time a system of these operations should be general enough to make possible representation of wide classes of systems. Usually operations of some abstract machine are used. Examples of such specifications are various kinds of automata: from finite state machines (FSMs) to Turing machines through communicating and/or extended FSMs (CFSMs, EFSMs, CEFSMs), labeled transition systems (LTSs), input/output state machines (IOSM) and transition systems (IOTS), Statecharts, timed automata, abstract state machines (ASMs), etc. Petri nets, CSP, CCS, SDL, Estelle, LOTOS, and other concurrency-related formalisms and languages based on operations performed by some virtual machine are also examples of this approach. Specifications in VDMor RSL [3] languages can also be written in an operational manner.

  The obvious and the most usual way to produce oracles from operational specifications is to execute them, obtain a result, and compare it with the result of the corresponding operation of the target system. The more details on test oracle construction on the base of ASM specifications can be found in [4,5].

  Operational models of complex systems are usually nondeterministic, since they are much more abstract than the corresponding systems. Wide nondeterminism is a serious obstacle for effective test construction. In the works of Software Engineering Foundations group in Microsoft Research [6] this problem is addressed by postponing the calculation of specification results until the implementation result is available. This approach solves the problem when the set of possible results is not too large, but it does not cover all possible systems. For example, when the target system solves the system of differential equations, in general we cannot adequately evaluate its behavior on the base of some other algorithm that solves such systems. This is because the bounded precision of calculations makes possible several locally 'correct' solutions (that can be distinguished with machine floating-point numbers) and the instability of the system may make them very different globally, so the number of distinguishable solutions becomes very large.

- *Contract specifications.* Software contracts [7,8] describe requirements to the system as contracts of its interface operations. Contract of an operation includes its precondition and postcondition. Precondition describes the obligations of the operation's clients, it should hold when one of them calls this operation. Postcondition describes the obligations of the system in response to call of this operation when its precondition holds. Usually types of arguments and results of interface operations also have contracts. Contract of a data type is formulated as a number of constraints on the constituents of an object of this type. Such constraints are called data integrity constraints or invariants. Contract specifications are supported by Z, B, VDM, RSL [3] , and Eiffel [9].

The way to construct oracles from contract specifications is quite obvious: we can check the precondition, then, if precondition holds, call the target operation and obtain its results, and then check the postcondition. Invariants can be checked as parts of both pre- and postconditions. The approaches similar to this one are used in [10,11,12]. One of the problems of the approach is concerned with storing the pre-state of the system, which can be used to check the postcondition after the operation execution. The other problem is description of active systems, which can provide actions without any external stimuli and may not terminate. Nontermination problem is addressed by LD-relations introduced by D. Parnas and used in [10].

- *Axiomatic specifications.* Axiomatic specifications describe the system as a set of functions with interrelated behavior. The interrelations are stated as a set of constraints on the combined behavior of several functions. These constraints can be formulated in first order logic or higher order theories. In the first case such specifications are usually called *algebraic.* Examples of languages used for algebraic specifications are OBJ, Larch, ML, and so on.

  Strictly speaking, other types of specifications can be considered as axioms of special kind. For example, contracts are also axioms of the kind: "when precondition holds before the operation call, then the postcondition holds after the call". But the structure of contracts and the techniques used to deal with them are so special, that they worth to be considered separately.

  Authors do not know on any use of axiomatic specifications other then algebraic ones (for, example, higher order theories) for test development. Algebraic specifications can be transformed into tests, but usually they can hardly be used to check the arbitrary possible call of system operations. An example of an approach for constructing general oracles from algebraic axioms is presented in [13]. This approach is based on rewriting terms into normal form and requires algebraic axiom set to be complete and confluent, that is, to guarantee the uniqueness of normal form for terms.

  The main problem of algebraic specifications is extreme difficulty of their construction for real-life systems. Specification developer should have advanced mathematical education and specific experience to deal with them in a useful manner.

- *History-based specifications.* History-based specifications define the set of possible sequences of actions related with the target system. This can be done with the help of some constraints on such sequences or by explicit presentation of parts of such sequences with some combination rules for them. History-based specifications include the approaches using various temporal logic formalisms, which makes possible to refer to past or future, trace specification methods, and scenario-based approaches, including methods based on MSC notation.

Examples of oracle construction methods based on temporal logics are given in the works [14,15,16]. Trace specification methods are presented in [17,18]. Test construction on the base on scenario-based specifications is dealt with in [19]. Oracles constructed from specifications of this kind usually evaluate some trace presenting the history of observable actions related with the target system. Usually, there are no ways to produce an oracle to evaluate system behavior in response to a single call.

From the one side, history-based methods are useful to provide a description of history-dependent behavior without introduction of auxiliary variables for storing history information, but from the other side they seem to be suitable only for a class of systems, which behavior depends on the history in a specific simple way, or can be constructed by composition from a small number of scenarios. In more complex cases such methods become too hard to use.

2.2 Test Quality Measures

For adequate measurement of testing quality both coverage of source code and coverage of requirements should be taken into account. Since we consider model based approaches here, implementation-based coverage metrics details are skipped. The interested reader may refer to [20] where the basic metrics are presented or to the general survey in [21].

Model based test coverage criteria should be considered in connection with method used to construct a model.

- *Operational specifications.* On the base of operational specifications we can obtain test coverage metrics if consider them as a program for abstract machine and construct coverage metrics as it were a source code.

  Coverage criteria based on EFSM-like models of the target system are considered in [22]. They are divided in control flow-based, such as transition coverage, transition-pair coverage, and data flow-based, as all-uses coverage or *du*-path coverage. Examples of such approaches based on ASM specifications are given in [23].

  The problem of coverage criteria based on operational specifications is their possible dependence on the structure of the modeling algorithms chosen. In general there is no guarantee that the algorithms actually used in the system have the similar structure. For example, we can calculate the square root function with the help of iteration process $x_n = (x_{n-1} + x/x_{n-1})/2$, as a sum of infinite row expansion of $(1 + (x - 1))^{1/2}$, or use an identity, such as $sqrt\,(x) = exp\,(ln\,(x)/2)$. Each of these methods gives its own coverage criteria, weakly related to each other. It is not easy to decide which one we should consider as the most natural one. And we need some implementation-specific information to choose one of them as more suitable.

- *Contract specifications.* Software contracts can be used to define coverage criteria if we separate different kinds of behavior described with different expressions in operation postcondition or try to apply usual code coverage metrics to pre- and postconditions considered as a code.

  Category-partition method [24] stays at the origin of most part of approaches using contract specifications for defining test coverage criteria. Examples of such approaches are given in [25,26].

  The coverage criteria constructed on the base of contract specifications can be too abstract and lack details actually significant for test quality measurement. So, in practice they usually need to be augmented with additional predicates related to the problem considered.

- *Axiomatic specifications.* Axiomatic specifications themselves provide coverage criteria based on the rules covered and states, which can be constructed as reduction of all possible histories. These metrics seem to be too coarse for practical usefulness. Some additional techniques should be used to obtain more useful measures for test quality. For example, we can construct a derived automaton model or use some combinational approaches. See the last item in the list on details.

- *History-based specifications.* History-based specifications give us only a set of possible traces and we can measure test coverage only on the base of actually executed traces. This is even less than we can extract from axioms. Again, we need to add some combinatoric considerations to achieve useful test adequacy criteria. See the last item on combinational approaches.

- *Probabilistic models.* Probabilistic usage models give a group of test quality metrics assessing not the degree of system features exploration during testing, but the degree of exploration of possible usage scenarios. Such models may serve to estimate the reliability of the system under test. They are usually represented as Markov chains describing the probability of external events depending on the history of previous system-environment interaction.

  Probabilistic models are used mostly as guides to the test data selection. They are presented, for example, in works [27,28,29].

- *Combinational approaches.* If we have a program model we usually can find various types of prime elements in it, which make up all the models of the same kind. For example, algebraic description of the system consists of a number of axioms, each being the equivalence of two sequences of operation calls. The elements we can see here — calls of operations, their sequences, and correspondence between such sequences. FSM-like models consists of states and transitions.

  We can introduce measures of test quality based on the complexity of possible combinations of prime elements, corresponding to the test executed. In case of algebraic specifications we can consider complexity of ground terms used for test construction and measure the test adequacy by the complexity of terms used. To make this measure more adequate to actual testing quality we may introduce an additional regularity hypothesis stating that it is sufficient to cover only terms of the complexity less than some fixed value [30,31].

The same approach may be used to measure test quality on the base of history-based specifications. We can try to calculate the percentage of bounded-length subsequences of the traces executed related to the total number of possible subsequences with the same bounds on their length.

We call such approaches combinational. They can help to introduce coverage criteria related with any kind of specifications used. Their practical usefulness for test adequacy measurement still needs to be investigated.

2.3 Methods of Test Sequence Construction

Methods of model based test sequence construction are considered here in connection with the kind of model used. In most cases they are targeted to obtain high or complete coverage according to some test coverage criterion.

- *Operational specifications.* Operation specifications in the form of automata of various kinds are the most widely used source for test sequence construction. The systematic exploration of test sequence construction methods based on FSMs are presented in [32,33,34]. [35] gives review of EFSM-based methods. The main idea of all these approaches is to construct a set of paths on the state-transition graph to obtain maximum coverage according to some coverage criterion — to cover all transitions, all adjacent transition pairs, all simple *du*-paths, and so on.

  The most part of methods based on (C)(E)FSM models provide construction of test sequence able to guarantee the conformance between a model and an implementation successfully tested in such a way. These methods and their derivatives are used to construct test on the base of formal specifications written in SDL [36], Estelle [37,38], LOTOS [39], or represented as Statecharts [40].

  Methods of test sequence construction based on ASM specifications of the target system are considered in [4,41]. The approach presented there is based on construction an FSM, which states correspond to combinations of values of prime logic formulas in branching conditions of ASM specification.

  FSMs provide the most suitable and deeply investigated methods of test sequence construction. Their main problem is state explosion — when a realistic system is considered, its FSM model adequate for sufficiently thorough testing appears to be huge and unmanageable. EFSMs and other kinds of state machines with possibility to extend states with data and transitions with parameters are more suitable to represent complex systems, but less convenient for analysis and test sequence generation. They present information on possible behaviors of the system in more implicit way and require deep analysis, which sometimes cannot be conducted automatically, to extract it in a form ready for test sequence generation.

- *Contract specifications.* Software contracts themselves cannot be used to construct test sequences. They should be transformed in or augmented by other models to become useful for this purposes.

  For example, Dick and Feivre [42] describe the method to construct FSM model from contract specifications and partition of the domain of each operation under test. The predicates defining the partition elements for some operations are considered in the parameters-states space. Each predicate corresponds to the set of pairs (parameters, state). The states of the resulting FSM are obtained as projections of such sets. The stimuli of this FSM correspond to the elements of the initial partitions for all operations. So, each operation gives rise to several stimuli. The transition tour on this FSM provides a complete coverage of the partitions chosen. The similar techniques are used in [43,44,45].
- *Axiomatic specifications.* Axiomatic specifications are rarely used for test sequence generation. There exist approaches that use their transformation into automata of some kind. A possible way to use axioms themselves to construct test sequences can be based on term rewriting. We can choose a sufficiently long seed sequence and try to rewrite it in as many ways as it sufficient for our testing needs or as it is possible due to our resources and budget, and then check the equivalence of system behaviour on the initial sequence and on any of the ones obtained by rewriting.
- *History-based specifications.* History-based specifications give the obvious method of test sequence construction based on the possible histories described. Usually specifications of this kind are transformed into some automaton to provide more convenient way of test sequence construction.

  Examples of test sequence construction methods using history-based specifications (temporal logic and MSC) are given in [46,47,48].
- *Other methods.* Other methods of test sequence construction include probabilistic and combinational approaches. They are usually designed to maximize some test adequacy metric. Such metrics are discussed in the previous subsection.

2.4 Problem of Changes in Test Development

The usual way used to solve the problems caused by continuous changes in requirements to the software, in development technologies used, in developer teams, or in the system's environment is abstraction. Abstract descriptions and specifications are more reusable. Stepwise refinement process helps to produce more detailed and specific specifications and code. Abstraction and refinement are also used to cope with changes in test development.

The first and the most obvious place for abstraction is the boundary between model and implementation. If the model used for test development has the same level of abstraction and the same details as the implementation, it does not provide an effective tool for testing, because requires to increase expenditures on the development in more than two times.

So, usually model is an abstraction of implementation. But then the question arises on the *conformance relation* between them, which is the main object of testing.

Designers of test development methods in academic community pay attention to strict definition of conformance relation checked by their methods. Examples of such relations for FSM-based approaches are given in [33]. In the works of J. Tretmans [49] *ioco*-relation is defined as conformance relation for LTS and IOLTS-based approaches to test construction.

To define conformance relation specification and implementation are considered as models of the same kind: both as FSMs, LTSs, and so on. The other characteristic of the existing definitions is one-to-one correspondence between observable events, i.e., inputs and outputs, in specification and in implementation. Both these factors may cause problems when the testing method is applied to real software. First, the software is required to be *implicitly* modeled in the metamodel used by the method, whatever it really means. Second, the observable events related with implementation usually should be abstracted to stay in correspondence with model events.

Such a situation leads to a shift in an abstraction level between a model used for test construction and the real software under test. This abstraction shift is not considered by the methods and usually processed in so called *system adapters,* which development is not regulated by any explicit rules.

One may object that formal rules for development of binding components between the formal model and informal implementation cannot exist. But some rules should be stated, lest the formally defined conformance relations will not be related with reality as soon as we begin to develop an adapter.

We need not only more strict rules of adapter construction, but also some extension of their possible functionality. We think that it is important to introduce in use more complicated conformance relations, for example, considering model as a factorization of implementation [50]. Such complication, although requires more strong hypotheses to enter in the area of formal testing, allows more simple models and more simple methods of test construction for complex systems, which appears to be rather useful in real practice.

The other important place for changes related with model-based testing is changes in requirements. Here systematic use of abstraction should lead to development of systematic formal models of various problem domains. The development of such models is now conducted in the context of MDA initiative of OMG [51]. Existence of such models poses another problem for designers of specification languages — they should introduce more flexible and powerful mechanisms of reuse. Such mechanisms should help to develop more flexible and reusable test suites based on more abstract models.

2.5 General Review Results

The review presented above shows that all techniques of formal description of system functionality have their advantages and drawbacks. None of them can be easily recognized as the most convenient for use in test development.

FSM-based models are very simple and comprehensible for developers, can be easily used for test sequence generation, but are too simplistic for software of real-life complexity. They can be made more scalable by introduction of data and parameters of transitions, but then they loose their suitability for analysis and still not obtain the generality and flexibility of more implicit descriptions, which are more suitable for construction of general-purpose test oracles.

On the other side, algebraic specifications, being the most implicit ones, are too difficult for use in real life development. They also lack suitability for construction of reasonable test adequacy metrics.

The solution seems to lie in integration of different techniques to use any of them in the domains where it is more suitable. We already have mentioned use of FSM models constructed on the base of contract specifications, or introduction of combinatoric considerations to measure test adequacy on the base of algebraic specifications. The next section provides a detailed presentation of UniTesK technology as an example of such integrated approach to test development.

3 UniTesK as a Symbiotic Solution

UniTesK [45,52,53] test development technology is designed by RedVesrt [1] group of ISP RAS to be used in industrial testing of complex software systems. It is based on the experience obtained by the group during several industrial software testing projects including operating systems for telecommunication switches, protocol implementations, P2P messaging management system, and compilers for C and FORTRAN languages.

The technology uses methods of test construction on the base of formal models of target software. But to make this process more effective, models of different kinds are used on different phases of development. The technology includes methods to construct one model on the base of others, thus seamlessly integrating them in the test development process. The main features of the technology, underlying test development process, and test architecture are described in the articles [45,52]. This section presents in more details the models used in UniTesK test development, relations between them, and reasons for choosing the presented solutions.

The functional requirements to the system behavior are represented as contract specifications. Contracts were chosen as the specifications that can be made either abstract or specific according to the goals of modeling. So, they do not impose too much determinism where it is not needed and are not ambiguous in necessary details. They are well-structured and close enough to requirements, and so do not lead to much effort spending for their development in real situations. In addition they are very suitable for automatic transformation into test oracles.

In the next sections we present a formal definition of language-independent part of contract specifications used in UniTesK. Then, we define formally the conformance relation between contract specifications and a target system represented as typed ASM. Since this relation in practically significant cases requires too much work to be checked, we introduce a hypothesis on equivalence of target system behavior in certain situations. Good candidates for such situations are given by natural partitioning of operation domains to subdomains of behavior described by one and the same constraint expression in postconditions. After that we give a method to construct an automata model based on the specifications given and set of predicates describing subdomains of equivalent behavior. When the behavior equivalence hypothesis holds and the target system obeys some reasonable restrictions, the tests constructed as a transition tour on this automaton are exhaustive, that is their successful execution proves the desired conformance between specifications and the system.

3.1 Typed Contract Specifications

A *many-sorted signature* $\Sigma = (S, F)$ consists of a set S of *sorts* and a set of *function symbols* $F = \bigcup_{\mathbf{w} \in S^*, s \in S} F_{\mathbf{w};s}$ layered by *function profiles* $(\mathbf{w}; s)$ consisting of a sequence of *argument sorts* $\mathbf{w} \in S^*$ and a *result sort* $s \in S$.

We often suppose that S includes sort **bool** and that F includes symbols of Boolean constants $\mathbf{t}, \mathbf{f} \in F_{\Lambda;\mathbf{bool}}$, functions $\neg \in F_{\mathbf{bool};\mathbf{bool}}$ and $\wedge, \vee \in F_{\mathbf{bool},\mathbf{bool};\mathbf{bool}}$, and equation symbol $=_s \in F_{s,s;\mathbf{bool}}$ for any sort s.

Then, we can define *sort terms* $\mathcal{T}^\Sigma$ and *function terms* or simply *terms* $\mathcal{F}^\Sigma$ on a signature $\Sigma = (S, F)$ in the following inductive manner.

- Sorts are sort terms and function symbols with a profile where the sequence of argument sorts is empty are terms.

$$S \subseteq \mathcal{T}^\Sigma \quad \text{and} \quad \forall s \in S \quad F_{\Lambda;s} \subseteq \mathcal{F}^\Sigma_{\Lambda;s}$$

- For each profile we have variables v_i, which are terms of this profile.
- We obtain a sort term $s_1 \times s_2$ if we apply *a product operation* to sort terms s_1 and s_2.

$$\forall s_1, s_2 \in \mathcal{T}^\Sigma \quad s_1 \times s_2 \in \mathcal{T}^\Sigma$$

- We obtain a term if we apply a function symbol to an array of terms with corresponding profiles.

$$\forall f \in F_{(s_1,\ldots,s_k);s},\ t_1 \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1l_1};s_1},\ldots,t_k \in \mathcal{F}^{\Sigma}_{s_{k1},\ldots,s_{kl_k};s_k}$$
$$f(t_1,\ldots,t_k) \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1l_k},\ldots,s_{k1},\ldots,s_{kl_k};s}$$

- We obtain terms if we take a pair of terms or a projection of a term with corresponding profile

$$\forall t_1 \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1k};s_1}, t_2 \in \mathcal{F}^{\Sigma}_{s_{21},\ldots,s_{2l};s_2} \quad (t_1,t_2) \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1k},s_{21},\ldots,s_{2l};s_1\times s_2}$$

$$\forall t \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1k};s_1\times s_2} \quad \pi_1 t \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1k};s_1} \text{ and } \pi_2 t \in \mathcal{F}^{\Sigma}_{s_{11},\ldots,s_{1k};s_2}$$

- We obtain a sort term if we take a *subsort* of sort term s according to a term t with the profile $(s;\mathbf{bool})$.

$$\forall s \in \mathcal{T}^{\Sigma}, t \in \mathcal{F}^{\Sigma}_{s;\mathbf{bool}} \quad \{s : t\} \in \mathcal{T}^{\Sigma}$$

A many-sorted structure or simply structure A of a signature $\Sigma = (S, F)$ maps each sort s to *a career set* s^A and each function symbol $f \in F_{s_1,\ldots,s_k;s}$ to *a partial function* $f^A : |A|_{s_1} \times \ldots \times |A|_{s_k} \stackrel{\cdot}{\to} |A|_s$. If S includes **bool** we require $\mathbf{bool}^A$ to have precisely two elements and map symbols for Boolean functions into corresponding operations on these elements. Equation symbol for each sort should be mapped in function mapping equal elements of the sort's career set into the image of $\mathbf{t}$ symbol.

Model mapping $\cdot^A$ can be extended onto sort terms and closed terms (having no free variables) in an obvious way.

$$(s_1 \times s_2)^A = s_1^A \times s_2^A$$

$$s : t^A = \{x \in s^A : t(x) = \mathbf{t}\}$$
$$(f(t_1,\ldots,t_k))^A : x \mapsto f^A(t_1^A(x),\ldots,t_k^A(x))$$
$$(t_1,t_2)^A : x \mapsto (t_1^A(x), t_2^A(x))$$
$$(\pi_i t)^A : x \mapsto \pi_i(t^A(x))$$

Σ-formulae Φ^{Σ} on the signature Σ are the usual many-sorted first-order formulae built from atomic formulae using quantifications and logic connectives. *The atomic formulae* are applications of function symbols $f \in F_{\mathbf{w};\mathbf{bool}}$ to argument terms of appropriate sorts.

The value of a closed term, the definedness of a term on an array of arguments, the satisfaction $A \models \phi$ of a closed formula ϕ in a structure A is defined by induction on their structure. The application of $f \in F_{\mathbf{w};\mathbf{bool}}$ to a sequence of arguments holds in A iff the values of all argument terms are defined and f^A is defined on these values and is equal to $\mathbf{t}^A$.

A typed asynchronous contract specification consists of the following parts.

- *A basic signature* $\Sigma_b = (S_b, F_b)$ consisting of a set of basic sorts and basic function symbols.

- *A dynamic signature* Σ_d including a set S_d of dynamic sorts and a set F_d of dynamic function symbols. Dynamic function symbols may have mixed profiles including sorts of S_d and sort terms on Σ_b. Dynamic sorts and function symbols represent the state of the system under specification. Further we denote *the union signature* $\Sigma_b \cup \Sigma_d = (S_b \cup S_d, F_b \cup F_d)$ as Δ.
- *Initial state specification* I, which is a set of Δ-structures.
- *A procedure symbols set* $\Pi = \bigcup_{\mathbf{w} \in (\mathcal{T}^{\Delta})^*, s \in \mathcal{T}^{\Delta}} (\Pi_{\mathbf{w};s} \cup \Pi_{\mathbf{w}}) \cup \bigcup_{s \in \mathcal{T}^{\Delta}} \Pi^a_{\Lambda;s}$ consisting of a set of *procedure symbols,* which can have profiles $(\mathbf{w}; s)$ (procedures returning a value) or $(\mathbf{w})$ (procedures returning nothing) constructed from sort terms on the union signature. Procedure symbols from Π^a are intended to specify *asynchronous reactions* of the system.
- *Preconditions* and *postconditions* of all procedures. $\mathbf{pre} : \Pi \to \Phi^{\Delta}$ and $\mathbf{pre}(p)$ have the only free variables corresponding to the arguments of $p \in \Pi$. $\mathbf{post} : \Pi \to \Phi^{\Delta'}$, where Δ' is constructed below. $\mathbf{post}(p)$ have the only free variables corresponding to the arguments of p and its result.

  Sorts of Δ' contains all basic sorts from S_b and two copies of each dynamic sort from S_d (say, s and s'). Function symbols of Δ' include all basic function symbols, all dynamic function symbols, and for each dynamic function symbol f they also include a function symbol f' with the profile, where each dynamic sort s met in the profile of f is replaced by s'. The basic sorts in the profile of f' remain the same as for f. So, we can express in postcondition a constraint on both the state preceding the procedure call and the resulting state.

  The problem is to match corresponding elements of the dynamic sort in pre-state and post-state. This problem can be solved (see [54] for more detailed explanation and references) by adding *a tracking map* symbol $\mathbf{tm} : s \to s'$ to Δ' for each dynamic sort s. This map is intended to map an element to the same element in the post-state, if it is preserved in s, or to nothing, if the procedure call removed it from s.

  We can interpret formulae from $\Phi^{\Delta'}$ in two Δ-structures A and A' in the following way. Basic sorts and function symbols are interpreted as usual. Unprimed dynamic sorts and symbols are interpreted in A, and their primed counterparts – in A'. Symbols $\mathbf{tm}_s$ are interpreted as partial one-to-one mappings $\mathbf{tm}_s^{A,A'} : s^A \to (s')^{A'}$. Then, by usual induction we define satisfaction $(A, A') \models \phi$ of closed formula $\phi \in \Phi^{\Delta'}$ in pair (A, A').

If Π^a is empty in the previous definition we say about *typed contract specification.*

3.2 Conformance Relations

For a system represented as a typed ASM (see for example [55] for general definition of ASM and [56] for typed ASM) that has a signature extending Δ we can define conformance relation, which formalize the concept of conformance between the specification and the system. First, consider the specifications without asynchronous reactions. We need to interpret calls of procedures as transitions in our ASM. To do so, we should add Π to ASM sorts. Then, we may have in ASM signature, for example, an external function symbol **in**, which in each state says what procedure with what arguments was just called (it has no parameters and returns a sequence starting with procedure symbol from Π and ending with the sequence of values of argument sorts), and a symbol **out** having the parameter of sort Π and returning the value of result sort of the procedure called just before the current state.

A typed ASM $\mathcal{A}$ with static signature $\Sigma_b' \supseteq \Sigma_b$ and dynamic signature $\Sigma_d' \supseteq \Sigma_d \cup \{\mathbf{in}, \mathbf{out}\}$ *conforms* to the specification with basic signature Σ_b, dynamic signature Σ_d, procedure set Π, and possible initial states I, iff the following requirements are satisfied

- For each initial state of $\mathcal{A}$ it is isomorphic to some element of I when we forget about all additional symbols in $\mathcal{A}$ signature.
- For each achievable transition $A \mapsto A'$ of $\mathcal{A}$ (that is a transition between two states, first of which can be achieved by $\mathcal{A}$ transitions from some initial state of $\mathcal{A}$), such that in $A' \models \mathbf{in} = p(v_1, \ldots, v_k)$ and $A' \models \mathbf{out}(\mathbf{p}) = r$, the following implication holds $A \models \mathbf{pre}(p)(v_1, \ldots, v_k) \Rightarrow (A, A') \models \mathbf{post}(p)(v_1, \ldots, v_k, r)$. This means that the transition *is permitted by specification.*

The concept of conformance in situation where asynchronous reactions are permitted is more sophisticated and not elaborated yet into the fully formal definition. It is based on the novelty introduced by UniTesK method – *plain concurrency axiom.* If a system obeys this axiom its asynchronous reactions can be ordered in such a way that the corresponding sequence of transitions can be found in specification. To model such a behavior we need to introduce in an ASM additional function symbol $\mathbf{out}^a$, which returns an element of Π^a. Then, the system conforms to the specification if the achievable paths in its states starting from an initial state and controlled by calling procedures from Π (that is, after each step of this path **out** returns $p \in \Pi$ or $\mathbf{out}^a$ returns $p \in \Pi^a$) can be split into pieces, each of which has the following properties

- The sequence of procedures called on this piece (the symbols returned by **out** and $\mathbf{out}^A$ can be ordered in such a way that the corresponding sequence of transitions between specification states (Δ-structures) is permitted by specification.
- The corresponding specification path starts and ends in *stationary states* A_1, A_2, that for each $p \in \Pi^a$ $A_i \not\models \mathbf{pre}(p)$. And the ends of the piece itself are also stationary, that is no there are no transitions from them to other state, in which $\mathbf{out}^a$ is defined.

Now we have discussed the form of specifications of the system functionality and conformance relation. In general case and actually for almost any example of real (really used for some purposes) software system we can develop a specification in a described manner, but it is impossible to construct test suite that can guarantee after successful execution that the system conforms to specification. The problem is the complexity of real systems and the consequent complexity of their specifications, even if we specify only the constraints we really want to check.

To cope with this complexity we need other model to be used. But the contract specifications constructed should be in clear relation with that model, since they state the requirements to the system. To construct more appropriate model we need to introduce some *behavior equivalence hypotheses,* which can help to reduce the effort needed for testing to a manageable size.

If we consider the postconditions of procedures, they can be represented in *a normal form* $\mathbf{post}(p) = \bigwedge_i (\phi_i \Rightarrow \psi_i)$, where all formulae ϕ_i do not depend on post-state, that is do not include primed symbols and a variable corresponding to the procedure result. This gives us a partition of procedure domain defined by its precondition. Each part of this partition is called *a functional branch.* Such a partition is a natural tools to measure the coverage obtained during testing.

We also can consider more detailed partition corresponding to single disjuncts in the DNF of $\mathbf{pre}(p) \wedge \phi_i$. Corresponding coverage criterion is called *disjunct coverage* in UniTesK.

Functional branches or disjuncts naturally determine regions of similar behavior of the corresponding procedure. So, we may state a hypothesis that the system operation corresponding to the specified procedure behaves *uniformly with respect to errors* in these domains, that is if it has an error, this error causes it to work incorrectly (according to specifications) for all values of arguments falling into one of the regions.

We can state such a hypothesis for any set of predicates that determine partitions of all operation domains. In this case we call such a predicate set *suitable for testing.* The reason to start from functional branches and disjuncts is that they are close to the natural way of operation implementation and so the hypothesis is more likely to hold for them.

3.3 Constructing an Automata Model

Having a specific system (represented as typed ASM), contract specification, and a finite suitable for testing set of predicates, we can construct an automata-like model actually suitable for test development. First, for each predicate P corresponding to the procedure $p \in \Pi$ consider the set of system states where exist such arguments of p, for which P holds. Then, take all nonempty intersections of such sets for all the predicates. That gives us a finite family $\mathbb{S}_i$ of sets of system states, which can be called *generalized states.* Correspondingly, let us call the predicates we use *generalized symbols,* and let say that a generalized symbol P is *admissible* in a generalized state $\mathbb{S}$, iff there exist a state $A \in \mathbb{S}$ and a set of values $v_i \in s_i^A$ of argument sorts s_i in the career sets of these sorts in the structure such that $A \models P(v_1, \ldots, v_k)$.

By construction of generalized states, if a generalized symbol P is admissible in a generalized state $\mathbb{S}$, then for each $A \in \mathbb{S}$ there exist a set $v_i \in s_i^A$ such that $A \models P(v_1, \ldots, v_k)$. So, if we want to *apply* an admissible generalized symbol in a generalized state, that is to execute a procedure with an arguments satisfying it, we always can do it, whatever particular state we use to represent the generalized one.

So, we can consider an automaton with states $\mathbb{S}_i$ and input symbols P_j. An application of an input symbol can lead us other state. Note, that this state in general depends both on the specific state A used to apply the symbol and values of arguments chosen. But in many cases we can iteratively split the generalized states, starting from the initial one, to impose determinism. The paper [50] describes a procedure that always stops if in the beginning we have finite deterministic system and gives a deterministic FSM as a result.

The other possible way is to resolve (maybe, with many possible solutions) constraints stated in pre- and postconditions and split the generalized states constructed from all nonisomorphic Δ-structures on the base of the solutions obtained. In this case the system under test is not involved in the process. If the specification actually defines a finite deterministic system, which is often the case for real systems, the process also results in a deterministic FSM.

It can be easily proved that if we construct a transition tour on the resulting FSM, we will cover all the situations represented by generalized symbols (for example, functional branches), which are achievable in the system considered. Therefore, if we start from a set of symbols suitable for testing, a transition tour on the FSM will be a strong test.

More formal statement is the following. If the system under test is represented as a typed ASM, we have all the premises of the conformance definition in the previous section, the system also satisfies *the admissibility hypothesis* – the application of procedure is defined in the system since it is admitted by the precondition in the specification, and we construct an FSM from a suitable for testing set of predicates, then successful execution of the transition tour will guarantee the conformance between the system and the specification.

FSM models are also attractive, because they are very suitable for test sequence construction and have a lot of different test selection methods providing test development flexibility often needed in practice. Use of combined model – contracts for description of system functionality and FSM for test selection – makes possible rather efficient test development using the suitable aspects of both and escaping their problems. The one noticeable problem of the approach is that the procedure of FSM construction requires either understating of the work of the system under test or the possibility to resolve completely implict constraints to all possible solutions, which is not always achievable and cannot be automated.

The same procedure as the one described above can be applied to construct an input/output automaton from the specifications with asynchronous reactions. But in this case the author does not know the similar formal results.

4 Conclusion

The idea to use multi-paradigm models for test development can be illustrated on several examples of both commercial tools and tools developed in academic society.

Rational TestRealtime [57] uses contract specifications in combination with FSM models. AsmL Test tool [58] developed in Microsoft Research uses ASM specifications for behavior description, but includes both automatic FSM construction on the base of ASM and user-defined definition of observable properties, which make up structure of states for some EFSM model. Gotcha-TCBeans tool [59,60] developed in IBM Research uses EFSM-like behavior specifications written in Murphy language and test directives that rule the test construction process. TorX [61] and TGV [62] tools both are based on LTS models augmented with test purposes described in scenario language.

UniTesK also is based on integration of several formal techniques. Its specific is in the following.

- Target system behavior is described in contract specifications — preconditions, postconditions, and invariants. Asynchronous system reactions also have contracts.
- Test adequacy criteria are formulated by user in the form of predicates defining the areas of equivalent behavior. More detailed criteria are constructed automatically on the base of user definitions and structure of specifications.
- On the base of coverage criterion chosen as a goal of testing and behavior specifications an (IO)FSM is constructed in such a way that a transition tour on it gives complete coverage according to the criterion chosen. After that test sequences can be constructed by means of standard techniques applied to the resulting IOFSM.
- Implicit representation of (IO)FSM models used as test scenarios makes them more compact and flexible in comparison with traditional approaches.
- Concurrent behavior of the system is tested using the same specifications as are used to test its sequential behavior.
- Mediators used to bind specification and implementation can realize rather complex conformance relations providing powerful mechanism for extensive reuse of specifications and tests.

UniTesK technology shows that integration of various formal techniques appears to be quite effective in test development for industrial applications. The technology is supported by tools developed in ISP RAS. Now tools for testing Java [63], C/C++ software, and .Net components are developed. All of them are commercial products [64], free licenses available for educational organizations. The tool for testing C software was successfully used to test several implementations of IPv6 protocol. The report on one of these projects can be found in [65]. The full list of projects conducted using the technology can be found on RedVerst web page [1].

References

[1] http://www.ispras.ru/groups/rv/rv.html

[2] L. Baresi and M. Young. Test Oracles. Tech. Report CIS-TR-01-02. Available at http://www.cs.uoregon.edu/∼michal/pubs/oracles.html

[3] The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.

[4] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Testing with Abstract State Machines. In R. Moreno-Diaz and A. Quesada-Arencibia, eds., Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001), Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001, pp. 257–261.

[5] M. Barnett and W. Schulte. Contracts, Components, and their Runtime Verification on the .NET Platform. Technical Report TR-2001-56, Microsotf Research.

[6] M. Barnett, L. Nachmanson, and W. Schulte. Conformance Checking of Components Against Their Non-deterministic Specifications. Microsoft Research Technical Report, MSR-TR-2000-56.

[7] Bertrand Meyer. Applying 'Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992, pp. 40–51.

[8] Bertrand Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall, 1997.

[9] Bertrand Meyer. Eiffel: The Language. Prentice Hall, 1992.

[10] D. Peters and D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Transactions on Software Engineering, 24(3):161–173, 1998.

[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] M. Obayashi, H. Kubota, S. P. McCarron, and L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.opengroup.org/

[13] S. Antoy and R. G. Hamlet. Automatically checking an implementation against its formal specification. Software Engineering, 26(1):55–69, 2000.

[14] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In 2-nd ACM SIGSOFT Symposium Foundations of Software Engineering, p. 140–153, Dec 1994.

[15] L. K. Dillon and Y. S. Ramakrishna. Generating Oracles from Your Favorite Temporal Logic Specifications. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, vol. 21(6) of ACM Software Engineering Notes, pp. 106–117, ACM Press, October 1996.

[16] A. Bader, A. S. M. Sajeev, and S. Ramakrishnan. Testing Concurrency and Communication in Distributed Systems. In International Conference on High Performance Computing, Dec 1998.

[17] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In G. Bracchi and P. C. Lockemann, editors, 2nd Conf. on European Cooperation in Informatics on Information Systems Methodology, Lecture Notes in Computer Science 65, pp. 211–236, Venice, Italy, 1978. Springer-Verlag.

[18] M. Iglewski, J. Madey, and K. Stencel. On fundamentals of the trace assertion method. Technical Report RR 94/09-6, Université du Québec à Hull, Hull, Canada, 1994.

[19] J. Grabowski, D. Hogrefe, R. Nahm. Test case generation with test purpose specification by MSCs. In O. Faergemand and A. Sarma, editors, 6-th SDL Forum, pages 253–266, Darmstadt, Germany, North-Holland 1993.

[20] B. Beizer. Software Testing Techniques. van Nostrand Reinhold, 1990.

[21] H. Zhu, P. A. V. Hall, and J. H. R. MAY. Software Unit Test Coverage and Adequacy. ACM Computing Surveys, Vol. 29, No. 4, December 1997.

[22] J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-based Tests. Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99), pp. 119–131, Las Vegas, NV, October 1999.

[23] A. Gargantini and E. Riccobene. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence Generation. Journal of Universal Computer Science, 7(11):1051–1068, 2001.

[24] T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM, 31(6):676–686, June 1988.

[25] N. Amla and P. Ammann. Using Z Specifications in Category Partition Testing. In Proceedings of the Seventh Annual Conference on Computer Assurance (COMPASS'92), Gaithersburg MD, June 1992. IEEE Computer Society Press.

[26] P. Stocks and D. Carrington. Test Templates: A Specification-Based Testing Framework. In Proceedings of the 15-th International Conference on Software Engineering, pp. 405–414, Baltimore, MD, May 1993.

[27] J. A. Whittaker. Markov chain techniques for software testing and reliability analysis. Ph.D. dissertation, Dept. of Comput. Sci., Univ. of Tennessee, Knoxville, USA, 1992.

[28] J. A. Whittaker and M. G. Thomason. A Markov Chain Model for Statistical Software Testing. IEEE Transactions on Software Engineering. vol. 20. No. 10, October 1994, pp. 812–824.

[29] G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical Testing of Software Based on a Usage Model. Software Practice and Experience, January 1995, pp. 97–108.

[30] M.-C. Gaudel, and B. Marre. Algebraic specifications and software testing: Theory and application. In Rapport LRI 407. (Jan. 1988), pp. 124–133.

[31] L. Bouge, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. J. Syst. Softw. 6, 1986, pp. 343–360.

[32] D. Sidhu and T. Leung. Formals Methods for Protocols Testing: A Detailed Study. IEEE Transactions on Software Engineering, vol. 15. No. 4, 1989.

[33] G. v. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In Proceedings of ACM International Symposium on Software Testing and Analysis. Seattle, USA, 1994, pp. 109–123.

[34] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite-State Machines. A survey. Proceedings of the IEEE, Vol. 84, No. 8, 1996, pp. 1090–1123.

[35] C. Bourhfir, R. Dssouli, E. Aboulhamid. Automatic Test Generation for EFSM-based Systems. Technical report, IRO, Université de Montréal, 1996.

[36] C. Bourhfir, E. Aboulhamid, R. Dssouli, N. Rico. A test case generation approach for conformance testing of SDL systems. Computer Communications 24(3-4):319–333 (2001).

[37] W. Chun, P. D. Amer. Test case generation for protocols specified in Estelle. In J. Quemada, J. Mañas, and E. Vázquez, editors. Formal Description Techniques, III, Madrid, Spain, North-Holland 1990, pp. 191–206.

[38] C. J. Wang, M. T. Liu. Automatic test case generation for Estelle. In International Conference on Network Protocols, pages 225–232, San Francisco, CA, USA, 1993.

[39] P. Tripathy and B. Sarikaya. Test Generation from LOTOS Specifications. IEEE Transactions on Computers, vol. 40, No. 4, pp. 543–552, April 1991.

[40] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. Automatic Test Generation from Statecharts Using Model Checking. Proceedings of Workshop on Formal Approaches to Testing of Software, Aug 2001, pp. 15–30.

[41] W. Grieskamp, Yu. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. Microsoft Research Technical Report MSR-TR-2001-97, October 2001 (revised May 2002).

[42] J. Dick and A. Feivre. Automating the generation and sequencing of test cases from model-based specifications. Proc. of Formal Methods Europe (FME 93), Springer-Verlag LNCS 670, p.268-284. 1993.

[43] H. Singh, M. Conrad, G. Egger, and S. Sadeghipour. Test case design based on Z and the classification-tree method. First IEEE International Conference on Formal Engineering Methods, November 1997.

[44] L. Murray, D. Carrington, I. MacColl, J. McDonald, and P. Strooper. Formal derivation of finite state machines for class testing. Z User Meeting (ZUM98), 1998. Also SVRC TR98-03.

[45] V. Kuliamin, A. Petrenko, A. Kossatchev, and I. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. In proceedings of 1-st Europpean Conference on Model-Driven Software Engineering, Dec 2003.

[46] S. Ramakrishnan and J. McGregor. Modelling and Testing OO Distributed Systems with Temporal Logic Formalisms. In 18th International IASTED Conference Applied Informatics'2000, Innsbruck, Austria, 2000.

[47] J. Grabowski. SDL and MSC Based Test Case Generation — An Overall View of the SAMSTAG Method. Technical report, University of Berne, IAM-94-0005, 1994.

[48] I.S. Chung, H.S. Kim, H.S. Bae, and B.S. Lee. Testing of Concurrent Programs based on Message Sequence Charts. In International Symposium on Parallel and Distributed Software Engineering (PDSET99), 1999.

[49] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. In T. Margaria and B. Stefen, editors, LNCS 1055, pp. 127–146. 2nd International workshop TACAS'96, 1996.

[50] I. Burdonov, A. Kossatchev, and V. Kulyamin. Application of finite automatons for program testing. Programming and Computer Software, 26(2):61–73, 2000.

[51] http://www.omg.org/mda/

[52] V. Kuliamin, A. Petrenko, I. Bourdonov, and A. Kossatchev. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.

[53] http://unitesk.ispras.ru

[54] H. Baumeister, A. Zamulin. State-Based Extension of CASL. Proceedings of IFM'2000.

[55] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In Specification and Validation Methods, E. Börger ed., Oxford University Press, 1995, pp. 9–36.

[56] A. Zamulin. Dynamic system specification by Typed Gurevich Machines. In Proc. Int. Conf. on Systems Science, Wroclaw, Poland, Sept. 15-18 1998.

[57] http://www.rational.com

[58] http://research.microsoft.com/fse/asml/

[59] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. IBM Systems Journal, volume 41, Number 1, 2002, pp. 89–110.

[60] http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html

[61] J. Tretmans, A. Belinfante. Automatic testing with formal methods. In EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis and Review, Barcelona, Spain, November 8-12, 1999. EuroStar Conferences, Galway, Ireland. Also: Technical Report TRCTIT-17, Centre for Telematics and Information Technology, University of Twente, The Netherlands.

[62] J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. In Special Issue on Industrially Relevant Applications of Formal Analysis Techniques, J. F. Groote and M. Rem, editors, Elsevier Science publisher, 1996.

[63] V. V. Kuliamin, A. K. Petrenko, I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. S. Kossatchev, and S. V. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'01. LNCS 2244, pp. 301–307. Springer-Verlag, 2001.

[64] http://www.atssoft.com

[65] http://www.ispras.ru/∼RedVerst/RedVerst/White Papers/MSRIPv6 Verification Project/Main.html