Programming and Computer Software, Vol. 29, No. 6, 2003, pp. 310–322. Translated from Programmirovanie, Vol. 29, No. 6, 2003.

The UniTesK Approach to Designing Test Suites

V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov

Institute for System Programming, Russian Academy of Sciences, ul. Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia

E-mail: igor@ispras.ru; kos@ispras.ru; kuliamin@ispras.ru

Received June 9, 2003

Abstract—Principles of the UniTesK test development technology based on the use of formal models of target software are presented. This technology was developed by the RedVerst group in the Institute for System Programming, Russian Academy of Sciences (ISPRAS) [1], which obtained rich experience in testing and verification of complex commercial software.

1. INTRODUCTION

At the present time, computer software production is so huge and the software is so complicated that the need for commercial testing technologies has become evident. The development of technologies that could ensure high-quality and methodical testing of target software as well as automated test suite development has become urgent. The conventional test development by hand has become inappropriate for testing large complex software systems.

Testing automation is usually reduced to the automatic execution of a series of tests and generation a report. It is more difficult to automate test generation and analysis of the results; this is because these tasks require that software specifications be available. However, the latter are often represented informally or even exist only in the form of knowledge and expertise of analysts, designers, and software developers.

In order to involve informal specifications in automated test suite development, they must be formalized to make them understandable for computer. In other words, a *formal specification* of the target system must be designed. Then, this specification can be transformed into a program that will be able to check if the software conforms to the specification.

Although test generation methods based on formal specifications are actively developed in the academic community, only a small fraction of these methods can be used in commercial software production. The main problem is that commercial software production needs technologies rather than separate methods; i.e., it needs a system of methods for performing a set of interrelated tasks concerning testing; moreover, these methods must be integrated into a development environment.

In this paper, we describe the UniTesK technology, which was developed in the ISPRAS on the basis of the experience obtained in the course of testing several complex commercial software systems. This technology is aimed at enabling commercial software developers to use cutting-edge methods of software testing. The main purpose of UniTesK is the development of functional tests based on models of functional requirements for the target system. Problems of designing tests for the verification nonfunctional requirements are beyond the scope of this paper.

The paper is organized as follows. Section 2 describes the basic elements of the UniTesK technology; first, basic concepts are outlined, and then some of them are presented in detail. In Section 3, UniTesK is compared with some other model-based approaches to test development. The final section gives examples of UniTesK applications and outlines directions of the further development of this technology.

2. UNITESK TECHNOLOGY

2.1. Basic Principles of UniTesK

We believe that a test development technology for the verification of general-purpose software can be widely used in commercial projects only if it has the following properties. First, all operations defined in the system must be (if possible) supported by corresponding tools. Second, it must provide a wide variety of features so that it can be used for testing various software in different subject areas. Finally, it is important that this technology could be integrated with available development processes; in particular, it must be based on widely used and simple concepts and notation so that it does not require long and expensive training of personnel.

To ensure these properties, the following principles were used in UniTesK.

• To ensure maximum flexibility, a universal test architecture was designed, which determines a set of test components with a distinctly separated functions and clear interfaces, so that a great variety of test types for various programs can be implemented in the framework of this architecture.

0361-7688/03/2906-0310$25.00

• In order to enable a significant degree of automation, the information that must be provided by the test developer was concentrated in a small number of components. All other components of the test are generated automatically or are used in all tests in an invariable form. In many cases, all variable components of the test, except for specifications that determine the software correctness criteria, can be generated interactively based on answers provided by a user to a series of clearly formulated questions.

• We used the well-known Design by Contract approach [2–4] as a metamodel for representing formal specifications. Program contracts consist of preconditions and postconditions of interface operations and of data type invariants. On the one hand, program contracts are convenient for designers and developers since they are closely related to the structure of the software; on the other hand, they stimulate efforts aimed at the creation of implementation-independent correctness criteria of the target system. However, the main advantage of program contracts is that they enable automatic construction of oracles [5–7] (which check if the behavior of the target system complies with the specifications) and test coverage criteria, which are close to requirements coverage criteria.

• It is practically impossible to provide a universal mechanism for constructing individual test actions (e.g., calls of functions with various sets of parameters) such that this mechanism is efficient both in terms of testing time and coverage. However, it is relatively simple to construct an iterator that iterates through a large set of values of a certain type. Tools supporting UniTesk contain libraries of basic iterators through values of simple types, which can be immediately used to generate test actions or can be assembled into more complex generators. To save testing time, test inputs (actions) can be filtered to eliminate the actions that do not enhance the test coverage. Filters are generated automatically from test coverage criteria (see the item after the next one).

• To automatically construct sequences of test inputs, we model the system under test as a finite state automaton. A test sequence is generated as a sequence of calls of target operations corresponding to a path in the graph of transitions of the automaton; for example, it can be a traversal of all transitions of the automaton. Since the finite automaton model is used only to generate a test sequence rather than verify the behavior of the entire system, which is performed by oracles, the automaton does not need to be completely defined. It is sufficient to specify a way for identifying its states and a method for iterating through inputs depending on the current state. Automata represented in such an implicit way can be conveniently represented in the form of test scenarios. Often, a test scenario can be generated automatically on the basis of a specification of target operations, a method for iterating through their parameters, and a testing strategy.

• The testing strategy must determine when the testing can be finished. In UniTesK, we suggest using the level of test coverage determined in accordance with a certain coverage criterion. One can automatically construct several test coverage criteria from functional specifications developed in accordance with the UniTesK technology. The user can control these criteria or define other criteria.

• To facilitate the integration with available software development processes, UniTesK can use extensions of popular programming languages to describe specifications and test scenarios. Classical formal specification languages can be used as well. The representation based on extensions of popular languages is clearer to many software developers; it makes it possible to reduce the training to one week. Immediately after training, a test developer can use UniTesK in practical work. In addition, the use of popular language extensions instead of a special language significantly facilitates the integration of the test and target systems, which is required for testing. Currently, the ISPRAS has developed UniTesK-compliant tools based on extensions of Java, C, and C#.

• Specifications based on program contracts can be used in the framework of UniTesK not only as insertions in the source code of the target system. They can be separated from the source code and used in the invariable form for testing different implementations of the same functionality. Thus, they can be considered as a formalization of functional requirements for the software. To define the relationship between the specifications and a particular implementation, special components called *mediators* are used. Mediators can perform fairly complicated transformations of interfaces. The use of mediators opens the following possibilities.

– Specifications can be much more abstract than the implementation; thus, they can be closer to the natural representation of functional requirements.

– Specifications remain the same for several versions of the target software. In order to revise a test suite so that it can be used for testing a new version in which only external interfaces have been changed (but their functionality remained the same), only mediators should be modified. In many cases, such a modification can be automated.

– An extensive reuse of specifications and tests becomes possible, which enhances the efficiency of effort involved in their development.

When the UniTesK technology is used in a specific domain, it often occurs that not all test construction techniques are required and not all components included in the universal test architecture must be constructed. Sometimes, the use of certain techniques is even impossible or requires too much effort. Then, specialized variants of the technology and supporting tools can be used.

For example, when compiler optimization blocks are tested, the development of a complete specification

for the functionality of such a block is very difficult; indeed, such a specification should describe the fact that the optimization of a program was performed. At the same time, it is relatively simple to compare the performance and identical functionality of special-type test programs. It is sufficient to execute these programs on a finite set of input data, which yields a method for constructing oracles; this method is not as general as that described above, but sufficient for practical purposes [8].

2.2. Universal Test Suite Architecture

The flexibility of a technology or tool, the possibility of using them in various situations and contexts is primarily determined by the architecture underlying this technology or tool. The test architecture used in UniTesK was designed on the basis of the experience obtained by testing complex commercial software. This architecture is designed for solving two basic problems.

• The construction of tests cannot be fully automated because only an expert can formulate the testing strategy and correctness criteria of the target software. Nevertheless, many things can be and should be automated.

• The architecture must combine uniformity with the capabilities of dealing with software from various subject areas.

The basic idea of the UniTesK test architecture is that a set of components is developed that can be used for testing various types of software following various test strategies. These components must have clearly defined responsibilities and interfaces through which they interact with each other. Information that can be provided only by the test designer is concentrated in a small number of components with neatly described roles. For each of these components, a simple and compact representation is developed that requires minimum effort from the user.

The UniTesK test architecture [9] is based on the following division of the test task into subtasks.

1. Verification of the system response to an isolated input.

2. Generation of an isolated test input.

3. Construction of a sequence of inputs aimed at achieving a desired coverage.

4. Establishing relationship between the test system based on an abstract model and the particular implementation of the target system.

To perform each of these subtasks, an appropriate support is provided.

In order to verify the software response to an isolated input, test oracles are used. Since the generation of test inputs is separated from the verification of the system response, one should know how to evaluate the behavior of the system in response to an arbitrary input action. The widely used method of generating oracles based on the computation of correct results for a fixed set of inputs is inappropriate for this purpose. We use general-type oracles based on predicates that depend on an input action and the system's response to this action.

Such oracles can be easily constructed from contract specifications (preconditions and postconditions of interface operations, and data type invariants that state data integrity conditions) [6, 7]. Under this approach, every input action is modeled as a call to an interface operation with a certain set of parameters, and the system response is modeled as the result of this call. Below, we will consider specific features of modeling asynchronous responses of the system and the corresponding specification techniques.

Isolated test inputs are constructed by iterating through operations and through a wide variety of parameter arrays for every fixed operation; the sets of parameters are filtered using the coverage criterion that is chosen as the aim of testing.

A set of isolated test inputs is insufficient for testing software with complicated behavior that depends on the preceding interaction of the system with its environment. In this case, series of test inputs are used, which are called *test sequences.* They are constructed so as to verify the behavior of the system in various situations determined by a sequence of preceding calls and system responses.

To construct test sequences, a finite-state model of the system is used. Such a model assumes that the dependence of the system's behavior on the history of its interaction with the environment can be reduced to the dependence on the current internal state of the system, which changes in response to call of the system, under the assumption that the set of attainable states is finite. Finite-state automata are simple, familiar to many developers, and can be used for modeling practically any program. To test concurrent or distributed systems, the variation of finite-state automata called *input/output automata* [10] is used. In these automata, transitions are labeled either by an input or output symbol. The resulting finite-state automaton is represented by an *test action iterator.* This component has an interface for obtaining the identifier of the current state, identifier of the next test input that is admissible in the current state, and for executing an action determined by its identifier.

A test sequence is generated dynamically in the course of testing by constructing an "exhausting" path trough the automaton transitions. This can be a traversal of all its states, all transitions, all pairs of adjacent transitions, and so on. An algorithm for the construction of such a path for a wide class of finite-state automata is implemented in a separate component called *test engine.*

A convenient (for a developer) description of the finite-state model used for testing is called *test scenario.* Test scenario provides the basis for generating the test action iterator. Scenarios can be constructed by

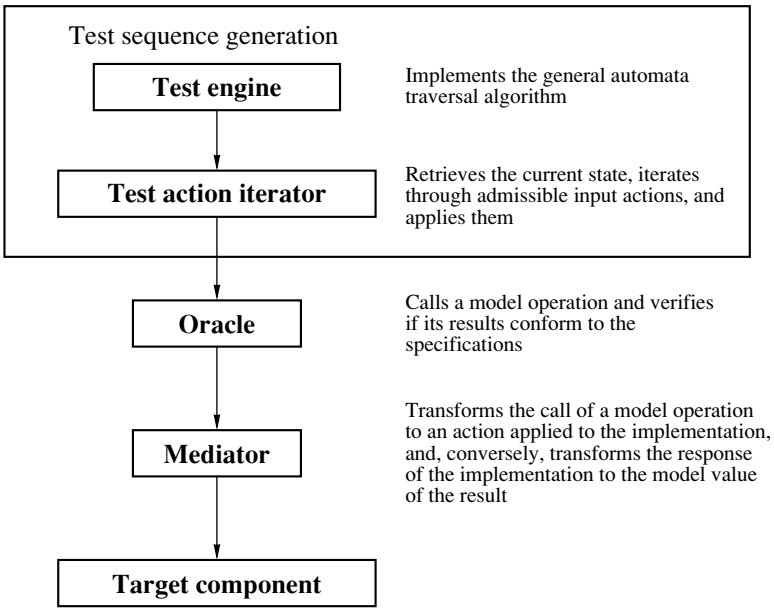

Fig. 1. Architecture of a UniTesK test suite.

hand, but in many cases they can be generated from specifications of operations, test coverage criterion, iterator through operation parameters, and a method for obtaining the identifier of the current state. Techniques for constructing test sequences are considered in Section 2.5 in more detail. Test engines of several types are provided in the form of library classes, and users do not need to develop them.

In order to use specifications written at a higher abstraction level than the target system, UniTesK provides the capability of using *mediators.* A mediator determines a relationship between a specification and an implementation of the corresponding functionality. Mediator also determines a transformation of model representations of inputs (calls of model operations) into the implementation representation and the inverse transformation of the target system responses into their model representation (the result returned by the model operation).

It is convenient to develop mediators in an extension of the target language, where only the transformations mentioned above should be described. Additional processing of the resulting code is required, since the mediator performs additional tasks related to specific features of the implementation environment and tracing the test progress. The code for performing these tasks is automatically added to the procedures transforming the test inputs and responses.

Figure 1 shows the main components of the test suite architecture used in UniTesK. In addition to these components, the test system includes some additional components responsible for tracing the test progress, synchronization of states between the model and implementation objects, and so on. These additional components are independent of the software under test and of the test strategy.

2.3. The Technique for Functional Requirements Description

UniTesk supports an automatic generation of test oracles from specifications represented in the form of contracts. With such a description of the target system functionality, it is modeled as a set of components each of which has several interface operations with certain parameters. The environment of the system can invoke interface operations and obtain the results. These results depend on the operation called, its parameters, and the history of the interaction of the system with its environment. Essential information on the history is modeled as the *internal state* of the target system components. Thus, the behavior of an operation generally depends on the internal state and can change it.

Every operation is described by *preconditions* and *postconditions.* A precondition determines the conditions under which the operation can be called. It is the environment (clients) of the component that is responsible for the precondition to be satisfied. One can say that the precondition describes the domain of the operation in the space of all states and set of its parameters. A postcondition imposes restrictions on the possible combinations of the starting state, parameters, operation results, and the resulting state; these restrictions must be satisfied if the precondition held when the operation was invoked.

An operation can have parameters of certain types. These types, types of fields of the model state, and the types of model components are called *interface types.* For all interface types, the structure of data is described,

which can have integrity constraints represented in the form of *invariants*. The data structure of the model components determines all possible model states of the system.

Program contracts were chosen as the basic specification technique since they are simple enough and can be applied to software in various subject areas. Contract specifications can be made abstract or very detailed as required. An ordinary software developer can be easily taught to understand and use contract specifications.

In addition, the structure of program contracts is close to the architecture of the target system, which makes them clear for developers. The internal content of the contracts is close to the requirements for the system. Thus, the representation of the requirements in the form of contracts is easy. On the other hand, the result is considerably different from the description of the algorithms used in the implementation, which prevents similar errors in the specification and implementation simultaneously.

Contract specifications are not the sole kind of specifications supported by UniTesK. One can also use executable specifications, which explicitly describe how the result of an operation is calculated and how the internal state of the component is changed. However, in this case, equivalence criteria between model and implementation results must be specified, since these results are not necessarily identical.

Often, axiomatic specifications cannot be directly transformed into oracles, which verify if the system behaves correctly in response to an arbitrary input action. Therefore, axiomatic specifications are used only as additional verification criteria for certain test sequences; they are also used to construct test scenarios.

2.4. Specification-based Test Coverage Criteria

The structure of contracts is used in UniTesK to determine test coverage criteria, which are necessary to evaluate the quality of testing from the viewpoint of the requirements for the target software. To enable the system to automatically generate such criteria, additional constraints are imposed on the structure of postconditions. More precisely, additional operators for determining *functionality branches* are introduced. These statements are placed by the user in a postcondition. A functionality branch corresponds to a subdomain in the domain of the operation such that the operation's behavior is "the same" in this subdomain. For definiteness, we may assume that the behavior is "the same" if the constraints imposed on the operation's results and on the change of state are described by the same expression in the postcondition for all points of the subdomain.

The graph of a postcondition's calculation control flow must have exactly one statement at every path from the entry to any exit, which determines the functionality branch. The part of the path to such a statement must not contain branch points that depend on the results of the operation. Then, every admissible call of the operation can be unambiguously associated with a functionality branch; therefore, the quality of testing the operation can be measured as the percentage of functionality branches covered by the test. Moreover, the functionality branch can be determined from the current state of the components and the set of operation's parameters without actually performing the operation; this makes it possible to construct a filter that eliminates sets of parameters that do not enhance the coverage.

Knowing branches of functionality in a postcondition, one can automatically extract more detailed coverage criteria based on the structure of branch points in preconditions and postconditions. The most detailed criterion of this type is disjunct coverage criterion, which is determined by all possible combinations of values of prime logical formulas used in these branch points. It is an analog of the MC/DC criterion [11] for the code coverage.

If testing is aimed at achieving high coverage in terms of disjuncts, some disjuncts can be unattainable due to implicit semantic relationships between logical formulas used (these problems are similar to those occurring when the MC/DC criterion is used). Such problems are resolved by using an explicit description of relationships in the form of tautologies, i.e., logical expressions constructed from prime formulas that are identically true due to dependences between the values of formulas.

In addition to the possibility of controlling coverage criteria that are automatically extracted from the structure of specifications, the user can describe additional specification coverage criteria by hand as a set of predicates that depend on the parameters of operations and the state.

2.5. Generation of Test Sequences

UniTesK uses finite-state automaton models of the target software in the form of test scenarios to dynamically generate sequences of test inputs. A scenario determines what is considered as the automaton state and which operations (with their parameters) must be invoked in each state. In the course of a test execution, the test engine constructs an "exhaustive" path through the automaton's transitions thus generating a test sequence.

This method of test generation guarantees that the state of the system changes only due to invoking target operations and that only the states that are attainable in this way can appear in the course of testing. Thus, the iteration through states is performed automatically, and the test developer should only determine how to iterate through the parameters of operations invoked.

When the scenario is developed, a specification coverage criterion can be used as a target one, and a set of

states and transitions can be determined in such a way that the traversal of all transitions in the resulting automaton guarantees that the desired coverage is achieved. For this purpose, it is sufficient to consider a set of predicates that determines elements of the coverage for a certain operation under test as a set of domains in the space of states and parameters of this operation and project these domains on the space of states.

Having constructed all possible intersections of these projections for all operations under test, we obtain a collection of sets of states such that any operation invoked for two states from the same set covers the same elements with respect of the coverage criteria used. Therefore, all such states of the system are equivalent from the viewpoint of this coverage criterion, and one can consider those sets of states of the system as the states of the resulting automaton. Stimuli in such an automaton are equivalence classes (with respect to the coverage criterion) of operation calls; i.e., they are the equivalence classes that cover the same element of the coverage. An additional transformation of the automaton may be needed to make it deterministic (for details, see [12]).

In the course of testing, one can use automatically generated filters that eliminate the sets of arguments that do not enhance the coverage. The availability of such filters makes it possible to save user's efforts required to calculate the parameters needed to achieve the desired coverage; instead, one can specify a large set of parameters that surely contains all necessary parameters. Thus, UniTesK enables one to perform efficient testing aimed at high coverage level.

A test scenario is actually a finite automaton represented in an implicit form; i.e., states and transitions are not listed explicitly and final states of the transitions are not specified. Instead, one specifies a method for calculating the current state, a method for comparing states, a method for iterating through admissible inputs (operations under test and their parameters), which depends on the state, and a procedure used to apply the input action. Although such a representation of automaton models is uncommon, it makes it possible to describe complex models in a compact form and easily modify them.

A scenario can define the states of the automaton model not only on the basis of the model state described in the specification, but also take into account certain implementation aspects that are not described in the specifications. On the other hand, one can abstract oneself from certain details in the specifications, thus decreasing the number of states in the resulting model (see [12]). Thus, the test generation can vary independently of specifications and, therefore, independently of the mechanism used to verify the behavior under an individual input action.

Test scenarios can be developed by hand, but they can be also generated automatically with the help of an interactive tool called the *scenario generation template*, which requests the user to provide only the necessary information and uses reasonable default assumptions. The scenario generation template helps construct both scenarios that do not use filtration of test inputs and scenarios aimed at high coverage level in terms of one of the criteria extracted from the specifications.

Test scenarios written in terms of specifications determine abstract tests that can be used for testing any system described by those specifications. Besides, scenarios can be reused with the help of the inheritance mechanism. An inherited scenario can override the state calculation procedure and override or enhance the set of test input actions that can be applied to the system at every state.

In order to test concurrent and distributed systems, UniTesK uses special-type test engines generating pairs, triples, and wider sets of concurrent input actions at every state, and slightly enhanced specifications. In addition to specifications of operations, which model input actions applied to the target system and its synchronous response to them, one can specify *asynchronous responses* of the system; each of them is represented in the form of an operation without parameters having preconditions and postconditions (see [71]).

Systems that satisfy the plain concurency axiom can be tested without asynchronous response specifications. This axiom states that the result of execution of any set of operations coincides with the execution of the same set of operations in a certain order. For systems that do not satisfy this axiom, additional "firings" corresponding to asynchronous responses or internal (i.e., not observable from the outside) changes of system's states can be introduced, so that the resulting model is plain.

Automaton models used for testing such systems are a generalization of input/output automata [10]. When a plain system is tested, the following method is used to verify its behavior. If the input actions processed by the system and its asynchronous responses can be fully ordered in such a way that, in the resulting sequence, the corresponding precondition is satisfied before every call of an operation or response and the corresponding postcondition is satisfied after the operation or response, then the behavior of the system is correct. Actually, this means that the observed behavior of the system does not contradict the specifications. If no such ordering can be constructed, then we conclude that a discrepancy between the system's behavior and the specifications is discovered.

In addition to the capabilities mentioned above, UniTesK test scenarios make it possible to perform testing based on conventional scenarios. The latter assumes that sequences of inputs are generated by user-defined rules, and the user specifies a method for verifying the system behavior. In particular, one can use scenarios that refine the target system use cases.

Another method of constructing test scenarios is based on axiomatic specifications, which describe the

correct system behavior in the form of constraints imposed on the results of invoking certain sequences of target operations. Every such sequence with the verification of the corresponding constraints can be represented in the test scenario as an action that will be applied to all states where it is admissible. Algebraic axioms, which require that the results be equivalent for two or more calling sequences, can be verified by representing each sequence as a separate action and comparing the results with the results of executing other sequences for the same state of the system. Test scenarios provide a convenient mechanism for storing intermediate data (in this case, the results of executing preceding sequences) associated with the state identifier.

2.6. Definition of Relationships between the Specifications and Implementation

Specifications used in UniTesK for test development can be related to the implementation through mediators rather than directly. This makes it possible to develop more abstract specifications, which can be extracted from requirements and used for testing several versions of the target software. This makes test suites more abstract and reusable. In addition to the advantages mentioned above, there are additional advantages of using mediators:

• The correspondence between the requirements (represented by specifications) and test suites can be controlled automatically.

• The simultaneous development of the software and tests for it is supported, which reduces the development time while ensuring a certain quality level.

• A more efficient infrastructure for distributing commercial software components can be supported. One can develop open specifications for the functional capabilities of such components supplemented with a test suite, which demonstrates that the component actually implements the desired functions. The component developer can attach mediators to the implementation that relate it to the open specifications. Thus, every user or independent tester can verify the correctness of the implementation. In addition, users of such components can use extended test suites to verify the components as required.

Mediators can be developed by hand and thus determine nontrivial transformations between the model's interface and the implementation's interface. In simple cases, the *mediator generation template* can be used for automatic generation of mediators. In this case, the specification and implementation components that must be connected should be specified and a correspondence between their operations must be established. In this case, one must specify a method for transforming the model parameters of each operation into the implementation ones and a method for transforming the implementation results into the model ones if they are not identical.

UniTesK makes it possible to use externally visible information about the implementation state to construct the model state. The testing method assuming that the model state is constructed on the basis of available reliable information about the implementation state independently of target operations invoked is called the *open state testing*. When this kind of testing is used, the procedure for constructing the model state is implemented as a special operation in the mediator, which is automatically called by the test system upon every call of the target operation (if there are no concurrent calls of the target system and asynchronous responses).

If information for the construction of the model state is insufficient (or when concurrent calls are tested or when the system can give asynchronous responses), the *hidden state testing* is used. In this case, the model state obtained after the application of an operation is constructed on the basis of the model state preceding the operation call, parameters, and the results of the call. This method yields a hypothetical next model state under the condition that the observed results of the call do not contradict the specifications. The method is correct if the constraints specified in the postcondition of any operation can be uniquely resolved with respect to the model state after the application of this operation. Mediators for this kind of testing must know how to construct, for each model operation, the model state obtained after the application of this operation.

Figure 2 presents the general test development scheme following the UniTesK technology. Upon the specifications, scenarios, and mediators have been developed, they are compiled into oracles, test action iterators, and mediators, respectively, in the language of the target system; then, they are integrated into the final test program.

2.7. A Uniform Extension of Programming Languages

As a rule, formal specifications are written in specialized languages that have a large set of expressive means and rigorously defined semantics. UniTesK enables one to use such languages if, for any pair (specification language, implementation language), clear rules for transforming interfaces are formulated and development tools for such transformations are available.

However, formal specification languages are difficult to be used for testing in spite of their advantages. This is because of difficulties arising in the definition of those transformations. The difficulties arise due to different paradigms underlying the specification and implementation languages, the absence in specification languages analogs of certain concepts that are widely used in implementation languages (e.g., pointers), differences in the semantics of simple data types, and so on. Therefore, the work requires large efforts of highly qualified experts who know both languages very well. The required training is very costly and takes a long time.

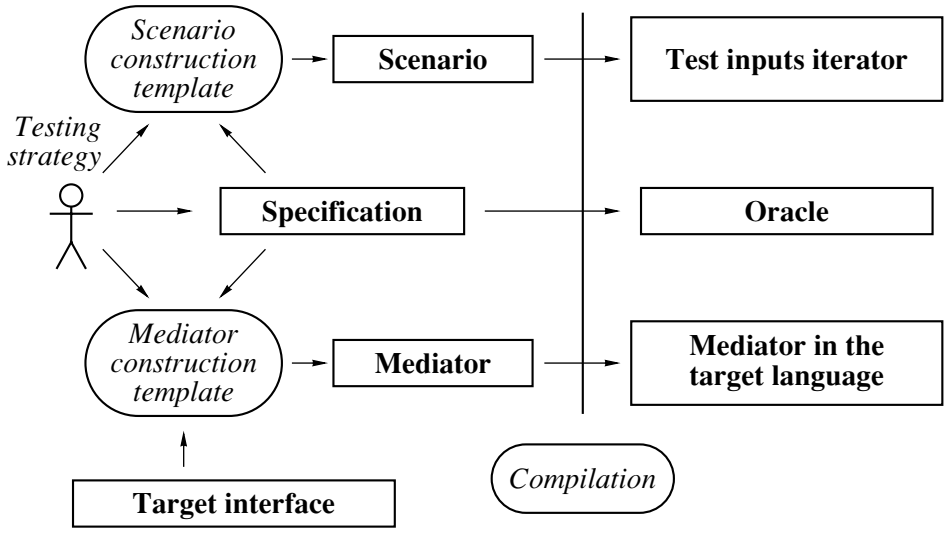

Fig. 2. The process of UniTesK test suite development.

In order to make the technology available to ordinary developers, UniTesK supports development of specifications and scenarios in extensions of popular programming languages. To this end, a unified system of basic concepts has been developed that are used for writing specifications and scenarios; these are the concepts of precondition, postcondition, invariant, branch of functionality, and the scenario method (which determines a uniform family of test inputs in the scenario). For each of these concepts, rules for the extension of the language by a corresponding construct are formulated. For languages that already have means for representing the corresponding concepts, only the lacking constructs are added.

An important advantage of using an extension of the target system language for writing specifications is that such specifications can be easier related to the implementation. An ordinary developer can easily teach himself to write specifications in the extension of the target language. The problem of insufficient expressive power of the majority of modern object-oriented languages is solved by using libraries of abstract types.

The problem of possible dependence of the specification meaning on the platform can be solved in several ways. First, one can prohibit using in specifications constructs that have insufficiently clear meaning and can be interpreted differently on different platforms. Second, libraries that operate identically on all supported platforms can be developed and recommended for use. Third, in specific cases, remote testing can be performed under which the test system is executed on the same platform where the specifications were developed.

2.8. Underlying Hypotheses and Formal Conclusions Based on Testing Results

When testing is performed according the UniTesK technology, the structure of the graph of automaton states is revealed (this automaton is implicitly described in the test scenario). The further actions are determined by the kind of testing, its aim, and the type of models employed on the basis of theoretical results on testing finite automata [13–15]. The basic result related with the simple case of checking if two finite deterministic automata (one of which represents the model and the other the implementation) coincide is as follows.

Testing is performed if the following assumptions are fulfilled.

1. The sets of stimuli and responses of the model and implementation automata coincide.

2. The sets of their states are subsets of the same set S.

3. The initial states are identical.

4. At every state that belongs to both automata, the sets of admissible stimuli are identical.

In the course of testing, it is verified that, for each model state attained, the implementation's response is admissible in the model for every stimulus that is admissible in the model. Moreover, if the model automaton is strongly connected and identity of states in both automata after each step is tested (open state testing), the model and implementation automata are identical if no errors were detected. More precisely, their subautomata that are attainable from the initial states are identical.

Note the difference between the open and hidden state testing. In the case of open state testing, we actually know the current implementation state and can check if it conforms to the model state. In this case, in order to conclude that all necessary transitions of the

implementation automaton have been checked, it is sufficient to traverse all transitions of the model automaton.

In the case of hidden state testing, we have no reliable information about the implementation state after every transition. To check all transitions in the implementation, it is not sufficient to traverse the model automaton. Therefore, more complicated testing methods must be used, for example, the method described in [13], which is based on characteristic sets of sequences, or the probabilistic method (see [13]).

If both the model and the implementation automatons are nondeterministic, an additional assumption is required that allows one to check whether nondeterministic transitions are correct. Two types of such assumptions can be used. For the first type, it is assumed that if the implementation automaton is affected many times by a stimulus, we obtain and, therefore, verify all possible transitions that can be fired by this stimulus (see, e.g. [16]). For the second type, it is assumed that the nondeterminism of the system is "homogeneous" with respect to errors; i.e., if a transition from a certain state in the model does not conform with the transition in the implementation for a certain stimulus, then there is a mismatch for all transitions from this state that can be fired by the same stimulus [17].

In addition, if the model and the implementation are nondeterministic, the requirement of strong connectivity of the model automaton must be replaced by the requirement of strong Δ-connectivity, which makes it possible to traverse the nondeterministic automaton (see [17] for details).

The above assumption that the states of the model and implementation automata belong to the same set or the equivalent assumption that there exists a one-to-one mapping between them can be rarely used in practice due to differences in the abstraction levels of the model and implementation. Generally, one can conclude that there is a correspondence between sequences of stimuli and responses in the model and implementation automata even if this correspondence cannot be naturally reduced to the correspondence between their states; however, the necessary theory has not yet been completed.

In a widespread particular case, the correspondence between the model and implementation can be represented in the form of factorization [12]. Then, there is a mapping $\varphi : S_I \longrightarrow S_M$ of the set of states of the implementation into the set of model states under which the preimage of a single model state can contain several elements. To enable testing, we need the assumption of "homogeneity" of the sets of admissible stimuli with respect to factorization; more precisely, the stimuli that are admissible in the model state $s \in S_M$ must be admissible in all corresponding implementation states $t \in \phi^{-1}s$. To conclude that the implementation conforms to the model, we need the assumption of "error homogeneity" with respect to factorization. More precisely, if, for a certain model state s and a model stimulus x that determines the transition to the model state s' with the response y, there exists a state $t \in \phi^{-1}s$ such that the transition to the state t fired by the stimulus x does not conform to the model (i.e., the response is not y or leads to the state t' such that $\phi(t') \neq s'$), then, for all $u \in \phi^{-1}s$, the transitions fired by the stimulus x from u do not conform to the model.

2.9. Execution of Tests and Analysis of Results

UniTesK tools support the automatic execution of tests developed with the help of UniTesK and the automatic collection of tracing information. After a test has been completed, a set of additional test reports can be generated based on this information. These reports show the structure of the automaton revealed in the course of testing, the level of test coverage for all test criteria defined for a certain specification operation, and information about detected violations due to errors in the target system or in specifications, scenarios, and mediators.

The test trace can be used to obtain additional information. For example, one can find out the kind of violation, the values of parameters of the operation that caused the violation, the particular constraint in the postcondition that was violated, and so on. The information contained in the trace and other reports is sufficient for debugging test system, for evaluation of the quality of testing, and often helps to localize errors.

3. COMPARISON WITH OTHER APPROACHES TO THE DEVELOPMENT OF MODEL-BASED TESTS

Although almost every feature of UniTesK can be found in other technologies and test development systems (sometimes, in a more advanced form), none of the approaches available in the academic community and software industry has all the capabilities of UniTesK.

In the short survey presented in this section, we focus on methods for test development supported by corresponding tools and aimed at the use in software industry. Thus, many interesting techniques are beyond the scope of this survey.

The available approaches to test development mainly rely on the conventional test architecture that set up several decades ago. In this architecture, a test suite is a set of test cases each of which is used to verify a certain property of the target system in a particular situation. In UniTesK, test suites are constructed in the form of scenarios, and every scenario actually plays the role of a set of test cases that verify the behavior of the target system when it calls a certain group of interfaces in various situations. As a result, the UniTesK test suite has more hierarchical levels, which is convenient when large and complex systems are tested. On the other hand, test cases make it possible to reconstruct certain

situations more efficiently when testing is repeated (for example, to check whether an error has been eliminated). Both schemes are equally suited for regression testing, since this usually requires the execution of the entire set of tests.

Automatic generation of test oracles differentiates UniTesK from such tools as JUnit [18], which automate only the execution of tests. However, automatic generation of test oracles is supported by many available tools, for example, the following ones.

• iContract [19, 20], JMSAssert [21], JML [22, 23], jContractor [24, 25], Jass [26, 27], Handshake [28], and JISL [29] use contract specifications written in the target system code in the form of comments in an extension of Java (reviews of such systems can be found in [30, 31]).

• SLIC [32] makes it possible to write contract specifications in an extension of C with the use of time logic predicates.

• Test RealTime [33] by Rational/IBM uses contracts and a description of the finite automaton model of the target component in the form of special scripts.

• JTest/JContract [34] by Parasoft and Korat [35] make it possible to write preconditions, postconditions, and invariants in the form of special comments in Java programs.

• ATG-Rover [36] uses specifications in the form of contracts written as comments in C, Java, or Verilog, which can include predicates of temporal logics LTL and MTL.

• The family of tools ADL [7] is based on extensions of C, C++, Java, and IDL, which are used for the development of contract specifications that are not strictly connected with a specific code.

• T-VEC [37] uses preconditions and postconditions in the form of tables in the SCR notation [38].

UniTesK differs from the three first tools above by substantial support of test development; in particular, it includes extraction of coverage criteria from specification and a mechanism for test sequence generation from scenarios. In JTest, the capability of automatic test sequence generation is declared, but the resulting sequences cannot contain more than three operation calls and are constructed randomly, so that the test sequence construction cannot be targeted to maximize test coverage.

Korat is one of the tools developed in the framework of the MulSaw project [39] in the laboratory of information science in MIT. This tool uses contracts written in JML for the generation of a set of input data for a single method in a Java class, including the object in which this method is called. The generated set guarantees the coverage of all logical branches in the specifications. Thus, instead of constructing a test sequence, one can immediately obtains the object in the desired state. On the other hand, specifications must be rigidly associated with an implementation. In particular, they should not allow states of the target component such that cannot occur in the course of its operation; otherwise, many generated test will correspond to unattainable states of the component.

ADL tools support test development only in the form of a library of input data generators, which is similar to the library of iterators in UniTesK. ATG-Rover can automatically generate templates of test sequences for specification coverage. It is not clear from the available documentation if these templates must be further processed by hand to become actual test sequences, but the possibility of such a processing is declared.

T-VEC uses specifications of a special form to automatically extract information on boundary values of the regions in which a function described by those specifications behaves "identically" (cf. the definitions of functionality branches in UniTesK). Test inputs are generated in such a way as to cover the boundary points of functionality branches for the function. A complete test suite is a list of pairs in which the first element is a set of parameters of the operation under test and the second element is the correct result of this operation on this set of parameters calculated from the specifications. Generation of test sequences is not supported.

To our knowledge, there are no other tools besides T-VEC that (as UniTesK) support generation of test suites aimed at high coverage in terms of criteria extracted from the internal structure of contract specifications. The majority of available tools can determine the coverage of specifications only as the percentage of operations that were called.

Generation of test sequences is supported by many tools that use models of the target system in the form of various automata such as extended finite automata, communicating finite automata, input/output automata, labeled transition systems, Petri nets, and so on. Such instruments suit well for the verification of telecommunication software, since formal specification languages based on the representations of software listed above (such as SDL [40–42], LOTOS [43], Estelle [44], ESTEREL [45, 46], or Lustre [47]) are often used in the development of such kind of software. The majority of these tools use a description of the system behavior in one of such languages as a specification, which is then transformed to an automaton model of the corresponding type.

In addition to the specifications of system behavior, some of these tools use a testing scenario usually called *test purpose*, which is formulated by the user in the form of a sequence of messages that are exchanged by software components (MSC) or in the form of a small automaton (see, e.g., [48–51]). Other tools use explicitly described automaton models for the generation of test sequences aimed at achieving a certain level of coverage with respect to a certain criterion [52–54]. As has been mentioned above, UniTesK supports generation of test sequences from user-defined scenarios and an

automaton model of the system; thus, UniTesK combines both approaches.

In terms of capabilities, GOTCHA-TCBeans [55, 56] (one of the test generation instruments joined in the AGEDIS project [57, 58]) and AsmL Test Tool [59, 60] are most close to UniTesK. Both these tools use automaton models of the target software. For GOTCHA-TCBeans, this model must be described in an extension of Murphi [61], while AsmL Test Tool uses a description of the target system in the form of an abstract state machine (ASM) as the specification [62, 63].

All three approaches use models of different types for constructing tests, which makes it possible to generate more efficient, flexible, and scalable tests; moreover, many components may be reused. In UniTesK, these are the model of system behavior represented by specifications and the testing model represented by the test scenario. In GOTCHA-TCBeans and other tools of the AGEDIS project, these are the automaton model of the system and a set of test directives that control the process of test generation. In recent versions of AsmL Test Tool, these are the ASM model of the system and a set of observable variables; the sets of values of these variables determine states of the automaton used to construct test sequence.

In those tools, certain techniques for decreasing the dimensionality of the model are used, which are similar to factorization in UniTesK. GOTCHA-TCBeans uses a particular case of factorization that ignores values of certain fields in the initial model state [64]. AsmL Test Tool can construct test sequences on the basis of a finite automaton whose states are obtained by reduction of the complete state of the initial automaton to the set of values of elementary logical formulas used in the description of this model's transitions [65].

The basic distinctions of UniTesK from GOTCHA-TCBeans and AsmL Test Tool are the support of programming language extensions used for writing specifications, the use of contract specifications, automatic control of specification coverage, and the use of filters aimed at producing test inputs that enhance coverage.

4. CONCLUSIONS

The UniTesK technology was developed on the basis of the experience gained when testing complex commercial software and the experience obtained in the integration of KVEST [6, 66], which is the predecessor of UniTesK, into the process of software development in Nortel Networks. This experience, and generally the experience of integration of technologies developed in academic institutions into software industry, shows that, for such a project to be successful, a technology must have a wide range of features and use concepts and notations that are familiar to ordinary software developers. Both these factors were taken into account in UniTesK.

The further development of UniTesK is supposed to follow several directions.

• Development of specialized tools for automated generation of mediators for popular types of component software.

• The scenario construction template will be extended so as to enable it to generate tests completely automatically only on the basis of the description of the interface to be tested.

• Automation of test suite development for software components that can call the environment and then use its answers (the so-called *inverse interface*).

• Integration into UniTesK testing methods for real-time systems and sophisticated approaches to testing concurrent and distributed systems.

Currently, in the Institute for System Programming, Russian Academy of Sciences (ISPRAS), in cooperation with Arithnet Technical Services [67], tools for constructing test suites in the framework of the UniTesK technology have been developed for software written in Java [68] and C#. For testing components written in C++, specifications and tests written in the extension of Java can be used; they can be binded with C++ code through mediators that are automatically generated from C++ header files containing the description of the interface under test. In addition, a similar tool CTesK has been developed for testing C programs [69].

These tools have been successfully used for testing software developed in ISPRAS, including the UniTesK support tools. CTesK was successfully used for testing several implementations of the IPv6 protocol [70]. On the basis of UniTesK principles, a specialized technology for testing compiler optimization blocks has been developed and tried out on commercial compilers produced by Intel [8]. A complete list of projects that are implemented with the use of the UniTesK technology can be found on the site of the RedVerst group of ISPRAS [1].

ACKNOWLEDGMENTS

This work was accomplished in the framework of the state contract no. 10002-251/p-21/019-027/101043-573 and supported by the Russian Foundation for Basic Research, project no. 02-01-00959.

REFERENCES

1. http://www.ispras.ru/groups/rv/rv.html.
2. Meyer, B., Applying "Design by Contract," *IEEE Computer*, 1992, vol. 25, no. 10, pp. 40–51.
3. Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1997, Second edition.
4. Meyer, B., *Eiffel: The Language*, Prentice Hall, 1992.
5. Peters, D. and Parnas, D. Using Test Oracles Generated from Program Documentation, *IEEE Transactions on Software Engineering*, 1998, vol. 24, no. 3, pp. 161–173.

6. Bourdonov, I., Kossatchev, A., Petrenko, A., and Gaiter, D., KVEST: Automated Generation of Test Suites from Formal Specifications, *Proc. of FM'99*, Lect. Notes Comput. Sci., Springer-Verlag, 1999, vol. 1708, pp. 608–621.
7. Obayashi, M., Kubota, H., McCarron, S.P., and Mallet, L., *The Assertion Based Testing Tool for OOP: ADL2*, http://adl.opengroup.org/.
8. Kossatchev, A., Petrenko, A., Zelenov, S., and Zelenova, S., Using Model-Based Approach for Automated Testing of Optimizing Compilers, *Proc. Int. Workshop on Program Understanding*, Gorno-Altaisk, 2003.
9. Bourdonov, L., Kossatchev, A., Kuliamin, V., and Petrenko, A., UniTesK Test Suite Architecture, *Proc. of FME 2002*, Lect. Notes Comput. Sci., Springer-Verlag, 2002, vol. 2391, pp. 77–88.
10. Zafiropulo, P., West, C.H., Rudin, H., Cowan, D.D., and Brand, D., Towards Analysing and Synthesizing Protocols, *IEEE Trans. Commun.*, 1980, vol. 28, no. 4, pp. 651–660.
11. Chilenski, J.J. and Miller, S.P., Applicability of modified condition/decision coverage to software testing, *Software Eng. J.*, 1994, September, pp. 193–200.
12. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Application of Finite Automata to Program Testing, *Programmirovanie*, 2000, no. 2, pp. 61–73.
13. Lee, D. and Yannakakis, M., Principles and Methods of Testing Finite-State Machines: A Survey, *Proc. IEEE*, 1996, vol. 84, no. 8, pp. 1090–1123.
14. von Bochmann, G. and Petrenko, A., Protocol Testing: Review of Methods and Relevance for Software Testing, *Proc. ACM Int. Symp. on Software Testing and Analysis*, Seattle, 1994, pp. 109–123.
15. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case, *Programmirovanie*, 2003, no. 5, pp. 11–30.
16. Fujiwara, S. and von Bochmann, G., Testing Nondeterministic Finite State Machine with Fault Coverage, Kroon, J., Heijink, R.J., and Brinksma, E., Eds., *Proc. IFIP TC6 Fourth Int. Workshop on Protocol Test Systems*, 1991, North-Holland, 1992, pp. 267–280.
17. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case, *Programmirovanie* (in press).
18. http://www.junit.org/index.htm.
19. Kramer, R., iContract – The Java Design by Contract Tool, *Proc. of TOOLS26: Technology of Object-Oriented Languages and Systems*, IEEE Computer Society, 1998, pp. 295–307.
20. http://www.reliable-systems.com/.
21. http://www.mmsindia.com/JMSAssert.html.
22. Bhorkar, A., A Run-time Assertion Checker for Java using JML, *Techn. Report 00-08*, Department of Computer Science, Iowa State University, 2000.
23. http://www.cs.iastate.edu/ leavens/JML.html.
24. Karaorman, M., Holzle, U., and Bruno, J., jContractor: A Reflective Java Library to Support Design by Contract, *Techn. Report TRCCS98-31*, University of California, Santa Barbara, Computer Science, January 19, 1999.
25. http://jcontractor.sourceforge.net/.
26. Bartetzko, D., Fisher, C., Moller, M., and Wehrheim, H., Jass—Java with Assertions, *Proc. of the First Workshop on Runtime Verification RV'O1*, Havelund, K. and Rosu, G., Eds., Electronic Notes in Theoretical Computer Science, Elsevier Science, 2001, vol. 55.
27. http://semantik.informatik.uni-oldenburg.de/~jass.
28. Duncan, A. and Hlzle, U., Adding Contracts to Java with Handshake, *Techn. Report TRCS98-32*, University of California, Santa Barbara, 1998.
29. Muller, P., Meyer, J., and Poetzsch-Heffter, A., Making Executable Interface Specifications More Expressive, *JIT'99 Java-Informations-Tage 1999*, Cap, C.H., Ed., Informatik Aktuell, Springer-Verlag, 1999.
30. Barneit, M. and Schulte, W., Contracts, Components, and Their Runtime Verification on the NET Platform, *Techn. Report TR-2001-56*, Microsoft Research.
31. Baresi, L. and Young, M., Test Oracles, *Techn. Report CISTR-01-02*, http://www.cs.uoregon.edu/~michal/pubs/oracles.html.
32. Ball, T. and Rajamani, S., SLIC: A Specification Language for Interface Checking (of C), *Techn. Report MSR-TR-2001-21*, Microsoft Research, 2002.
33. http://www.rational.com.
34. http://www.parasoft.com.
35. Boyapati, C., Khurshid, S., and Marinov, D., Korat: Automated Testing Based on Java Predicates, *Proc. of ISSTA 2002*, Rome, 2002.
36. http://www.time-rover.com.
37. http://www.t-vec.com.
38. Heitmeyer, C., Software Cost Reduction, *Encyclopedia of Software Engineering*, Marciniak, J.J., Ed., 2 vols., 2002.
39. http://mulsaw.lcs.mit.edu/.
40. Ellsberger, J., Hogrefe, D., and Sarma, A., *SDL—Formal Object-Oriented Language for Communicating Systems*, Prentice Hall, 1997.
41. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, Geneva: ITU-T, 1996.
42. *ITU-T Recommendation Z.100 Annex Fl: SDL formal definition—General*, Geneva: ITU-T, 2000.
43. *Information Processing Systems: Open Systems Interconnection—LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO/IEC 8807:1989, Geneva, 1989.
44. *ISO/TC97/SC21. Information Processing Systems: Open Systems Interconnection—Estelle—A Formal Description Technique Based on an Extended State Transition Model*, ISO 9074:1997, Geneva, 1997.
45. Berry, G., The Foundations of Esterel, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Plotkin, G., Stirling, C., and Tofte, M., Eds., MIT Press, 1998.
46. Boussinot, F. and de Simone, R., The Esterel Language, *Proc. IEEE*, 1991, vol. 79, pp. 1293–1304.
47. Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., The Synchronous Data Flow Programming Language LUSTRE, *Proc. IEEE*, 1991, vol. 79, pp. 1305–1320.
48. Grabowski, J., Hogrefe, D., and Nahm, R., Test case generation with test purpose specification by MSCs, *6th SDL Forum*, Faergemand, O. and Sarma, A., Eds., Darmstadt, Germany: North-Holland, 1993, pp. 253–266.

49. Wang, C.J. and Liu, M.T., Automatic Test Case Generation for Estelle, *Int. Conf. on Network Protocols*, San Francisco, 1993, pp. 225–232.
50. Caravel, H., Lang, F., and Mateescu, R., An overview of CADP 2001, *INRIA Techn. Report TR-254*, 2001.
51. Fernandez, J.-C., Jard, C., Jeron, T., and Viho, C., An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology, *Special Issue on Industrially Relevant Applications of Formal Analysis Techniques*, Groote, J.F. and Rem, M., Eds., Elsevier Science, 1996.
52. Tretmans, J. and Belinfante, A., Automatic Testing with Formal Methods, *EuroSTAR'99: 7th European Int. Conf. on Software Testing: Analysis and Review*, Barcelona, 1999, EuroStar Conferences, Galway, Ireland, Techn. Report TRCTIT-17, Centre for Telematics and Information Technology, University of Twente.
53. Bourhfir, C., Aboulhamid, E., Dssouli, R., and Rico, N., A Test Case Generation Approach for Conformance Testing of SDL Systems, *Computer Commun.*, 2001, vol. 24, nos. 3–4, pp. 319–333.
54. Chun, W. and Amer, P.D., Test Case Generation for Protocols Specified in Estelle, *Formal Description Techniques III*, Que-mada, J., Manas, J., and Vazquez, E., Eds., Madrid: North-Holland, 1990, pp. 191–206.
55. Farchi, E., Hartman, A., and Pinter, S.S., Using a Model-based Test Generator to Test for Standard Conformance, *IBM Syst. J.*, 2002, vol. 41, no. 1, pp. 89–110.
56. http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html.
57. Gronau, L., Hartmair, A., Klrshin, A., Nagin, K., and Olvovsky, S., A Methodology and Architecture for Automated Software Testing, http://www.haifa.il.ibm.com/projects/verification/gtcb/papers/gtcbmanda.pdf.
58. http://www.agedis.de/.
59. Grieskamp, W., Gurevich, Y., Schulte, W., and Veanes, M., Testing with Abstract State Machines, Moreno-Diaz, R. and Quesada-Arencibia, A., Eds., *Formal Methods and Tools for Computer Science (Proc. of Eurocast 2001)*, Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain, 2001, pp. 257–261.
60. http://research.microsoft.com/fse/asml/.
61. http://verify.stanford.edu/dill/murphi.html.
62. Gurevich, Y., Evolving Algebras: An Attempt to Discover Semantics, *Current Trends in Theoretical Computer Science*, Rozenberg, G. and Salomaa, A., Eds., World Scientific, 1993, pp. 266–292.
63. Borger, E. and Stark, R., *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.
64. Friedman, G., Hartman, A., Nagin, K., and Shiran, T., Projected State Machine Coverage for Software Testing, *Proc. of ISSTA 2002*, Rome, 2002.
65. Grieskamp, W., Gurevich, Y., Schulte, W., and Veanes, M., Generating Finite State Machines from Abstract State Machines, *Proc. of ISSTA'2002*, Techn. Report, MSRTR-2001-97, Microsoft Research,
66. http://www.frneurope.org/databases/fmadb088.html.
67. http://www.atssoft.com.
68. Bourdonov, I.B., Demakov, A.V., Jarov, A.A., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K., and Zelenov, S.V., Java Specification Extension for Automated Test Development, *Proc. of PSI'O1*, Lect. Notes Comput. Sci., Springer-Verlag, 2001, vol. 2244, pp. 301–307.
69. http://unitesk.ispras.ru.
70. http://www.ispras.ru/~RedVerst/RedVerst/White Papers/MSRIPv6 Verification Project/Main.html.
71. Kuliamin, V., Petrenko, A., Pakoulin, N., Kossatchev, A., and Bourdonov, I., Integration of Functional and Timed Testing of Real-time and Concurrent Systems, *Proc. of PSI'03*, Novosibirsk, 2003.