UniTesK Approach to Test Development: Achievements and Prospects

A. Barantsev, I. Burdonov, A. Demakov, S. Zelenov, A. Khoroshilov,
A. Kossatchev, V. Kuliamin, V. Omeltchenko, N. Pakoulin, A. Petrenko
{barancev, igor, demakov, zelenov, hed, kos, kuliamin, vitaliy, npak, petrenko}@ispras.ru

Abstract

The article expounds the basic principles of UniTesK test development technology based on formal models of target software. The article also presents the experience of application of this technology to various types of software, describes the problems arising during transition of this technology and other technologies based on formal methods to the industry, and gives possible solutions of these problems. UniTesK technology was developed by RedVerst group [RedVerst] of ISP RAS on the base of rich experience obtained in testing and verification of complex industrial software.
The article may be interested both for researchers in formal methods of software engineering and practitioners, who wish to know how can model based testing methods be applied to real-life large-scale software. The readers of the second group are advised to look through the sections “Basic UniTesK Principles”, “UniTesK Test Development Process”, proceed to the section “Experience of UniTesK Applications”, and then read other sections, which they may be interested in.

Introduction

Nowadays the software systems developed in the industry are so huge and so complex that the need for industrially applicable systematic testing technologies has become evident. The technologies that can guarantee both the qualitative and systematic testing and the high automation of the test development are especially urgent. The conventional test development methods based on manual work cannot provide testing quality for modern complex systems.

Testing automation is usually reduced to automatic test execution and test report generation. To automate test production and analysis of their results is much more difficult, because these tasks can be performed only on the base of *requirements* to the target software. However, the requirements are often represented informally or even exist only in the form of knowledge and experience of experts, analysts, and software designers.

To involve informal requirements in automated test development they must be formalized to enable computer to process them. In other words, *formal specifications* of the target system must be developed. These specifications can be transformed automatically into programs that are able to check the software conformance to the requirements specified.

Although test generation methods based on formal specifications or formal models are actively developed in the academic community, only several of these methods can be used in industrial software production. The main problem is that the industry needs *technology* rather than separate methods; i.e. it needs a system of methods to perform a set of interrelated tasks concerned with testing and test development; moreover, these methods must be supported by tools integrated into widely used development environments.

The article describes UniTesK technology developed in ISP RAS on the base of experience of several projects in verification and testing of complex industrial software. It is intended to introduce advanced test development methods in the software industry. The technology considered can be used to produce tests on the base of functional requirements to the target software. Nonfunctional testing and test development problems are out of the scope of this article.

The paper is organized as follows. The next section describes the principles of UniTesK technology starting form the general considerations and then presenting the details. The third section compares the technology with other model based test development with other approaches to model based test development. The fourth section briefly describes UniTesK case studies and the experience of technology transition to the industry. The conclusion considers the possible lines of future development.

UniTesK Technology

Basic UniTesK Principles

A test development technology for general-purpose software becomes applicable in the industrial software development only if it has the following properties. First, all activities defined by the technology should 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 problem domains, developed using different architectures with the help of various technologies. Finally, it should be integrated with usual 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.

The following solutions are suggested by UniTesK to ensure the properties states.

1. To provide the maximum flexibility *the universal test suite architecture* was developed. It defines the set of test components with clear interfaces and functions, so that a great variety of tests for various programs can be implemented in this framework.
2. To make test development as automated as possible all the information that can be provided only by the test developer is collected in a small number of components. The convenient notation is suggested to make possible short presentation of this information. All other test components are generated automatically form the ones mention above, or are used in all test in invariable form. In many cases all variable test components, except for specifications that define software correctness criteria, can be generated interactively on the base of user's answers on a series of clear questions.
3. The well-known *Design by Contract* approach [DBCA,DBCO,DBCE] is used to represent formal specifications. Program contracts consist of *preconditions* and *postconditions* of interface operations and data type *invariants*. On the one hand, program contracts are convenient for software developers since they are closely related to the architecture of the software; on the other hand, owing to their representation they stimulate efforts to formulate implementation-independent correctness criteria of the target system. The main advantage of program contracts is that they enable automatic construction of *test oracles* [Parnas,KVEST,ADLt] that check the conformance of the target system behavior to the specifications. Program contracts can be also used to create *test coverage criteria*, which are close to requirements coverage criteria, on the base of contract structure.
4. It is practically impossible to provide a universal mechanism for construction of single test actions (e.g., calls of functions with various sets of parameters) such that this mechanism is efficient enough both in terms of testing time and resulting test 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 providing values of simple types, which can be used to generate test actions themselves or can be assembled into more complex generators. To save testing time, test actions can be filtered to discard the actions that do not enhance the test coverage. Filters are generated automatically from test coverage criteria (see the item 6).
5. To automatically construct sequences of test inputs, we model the system under test as *a finite state machine* (FSM). A test sequence is generated as a sequence of calls of target operations corresponding to a path in the state transition graph of the FSM; for example, it can be a transition tour on the FSM. Since the FSM model is used only to generate a test sequence rather than to verify the behavior of the target system, which is performed by oracles, the FSM 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. FSMs 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 base of specifications of target operations, a method for iterating through their parameters, and a testing strategy.

6. The testing strategy determines when the testing can be finished. UniTesK uses the level of test coverage achieved according to some coverage criterion. One can automatically construct several test coverage criteria from specifications developed in accordance with the UniTesK technology. The user can control these criteria or define custom test coverage criteria.
7. To facilitate the integration with available software development processes, UniTesK can use extensions of widely used programming languages to represent specifications and test scenarios. Classical formal specification languages can be used as well. The representation based on extensions of programming languages is more clear to ordinary software developers; it makes it possible to reduce the training in the technology to one week. Immediately after training, a test developer can use UniTesK in practical work and produce useful results. In addition, the use of programming language extensions instead of a special language significantly facilitates the integration of the test and the target system, which is required for testing. Currently ISPRAS has developed UniTesK-compliant tools based on extensions of Java, C, and C#.
8. Specifications in the form of program contracts can be used in 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 to test different implementations of the same functionality. Thus, they can be considered as a formalization of functional requirements for the software. To provide the binding between 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. To revise a test suite for testing a new version in which only external interfaces have been changed and 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, often not all test construction techniques are required and not all components included in the universal test architecture should be constructed. Sometimes, the use of certain techniques is even impossible or requires too much effort. Nevertheless, specialized variants of the technology and supporting tools can be used.

For example, when compiler optimization modules are tested, the development of a complete specification for the functionality of such a module is very difficult, if ever possible; it should describe the fact that the optimization of a program has been actually performed. At the same time, it is relatively simple to compare the performance and functionality of special-type test programs after compilation with and without optimization. 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 [OPT].

Universal Test Suite Architecture

The flexibility of a technology or tool, the possibility of their use in various situations and contexts is primarily determined by the underlying architecture. The test architecture used in

UniTesK was designed on the base of the experience obtained by testing complex industrial software. This architecture is intended to solve the following two 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 capability to test software from various problem domains in projects having different goals.

The basic idea of the UniTesK test architecture is that a set of test components to develop 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 developer is collected 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 to create.

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

1. Verification of the system response to a single input action
2. Generation of a single test input action
3. Construction of a sequence of inputs intended to achieve a desired coverage
4. Establishing binding between the test system based on an abstract model and the particular implementation of the target system.

For each of these tasks an appropriate technological support is provided.

To verify the software response to a single input, *test oracles* are used. Since the generation of test inputs is separated from the verification of the system response, we 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 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 represented as preconditions and postconditions of interface operations, and data type invariants that state data integrity conditions [KVEST,ADLt]. 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 the specifics of modeling asynchronous system responses and the corresponding specification techniques.

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

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

To construct test sequences, an FSM 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 operations, under the assumption that the set of attainable states is finite. FSMs are simple, familiar to many developers, and can be used to model almost any program. To model concurrent or distributed systems, the variation of *input/output finite state machines* (IOFSMs) [10] are used. In these automata transitions are labeled either by an input or output symbol. The resulting automaton is represented by *a test action iterator* program component. This component has an interface to obtain the identifier of the current state, identifier of the next test input action that is admissible in the current state, and for executing an action determined by its identifier.

More details on automata models used to test concurrent systems can be found in [AsSM].

A test sequence is generated dynamically during test execution by constructing an "exhausting" path trough the automaton transitions. This can be a path covering 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 FSMs is implemented in a separate component called *test engine*.

An automaton model description convenient for a developer is called *test scenario*. Test scenario provides the basis for generating the test action iterator. Scenarios can be constructed by hand, but in many cases they can be generated from specifications of operations, test coverage criterion, iterators through operation' parameter arrays, and a method for obtaining the identifier of the current state. Techniques for constructing test sequences are considered in the section after the next one in more details. Test engines of several types are provided in the form of library classes, and users do not need to develop them.

To use specifications written at a higher abstraction level than the target system, *mediators* are used in UniTesK. A mediator determines a binding 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 their implementation representation and the inverse transformation of the target system responses into their model representation (the result returned by the model operation).

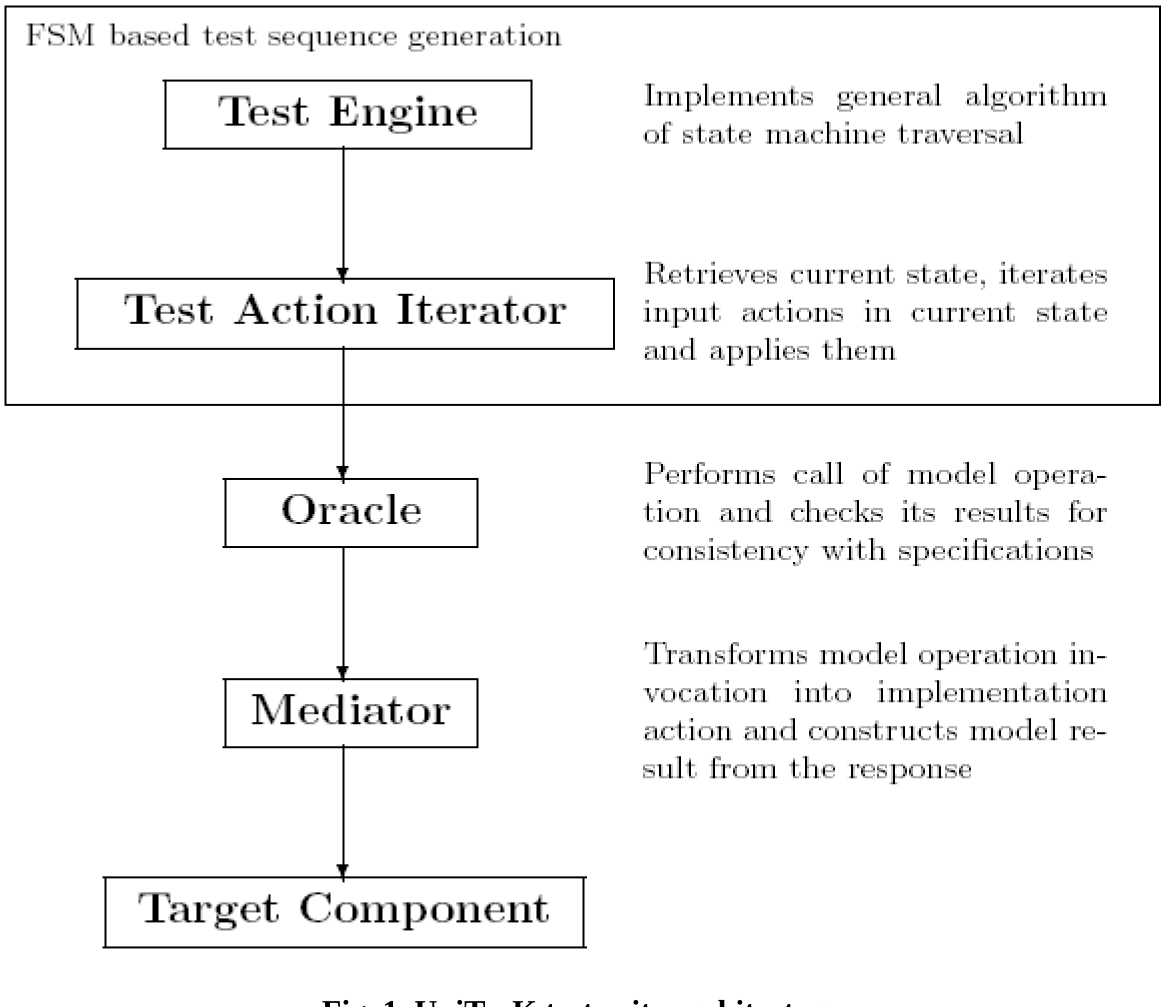

FSM based test sequence generation
Test Engine
Implements general algorithm of state machine traversal
Test Action Iterator
Retrieves current state, iterates input actions in current state and applies them
Oracle
Performs call of model operation and checks its results for consistency with specifications
Mediator
Transforms model operation invocation into implementation action and constructs model result from the response
Target Component

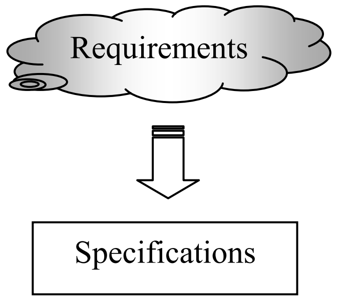
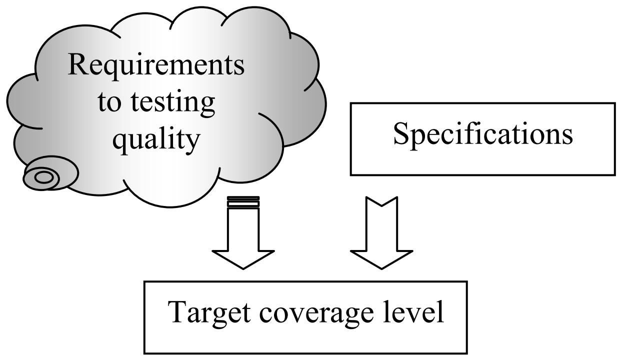
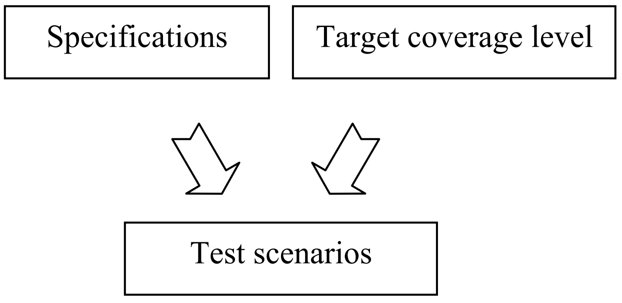
Fig. 1. UniTesK test suite architecture.

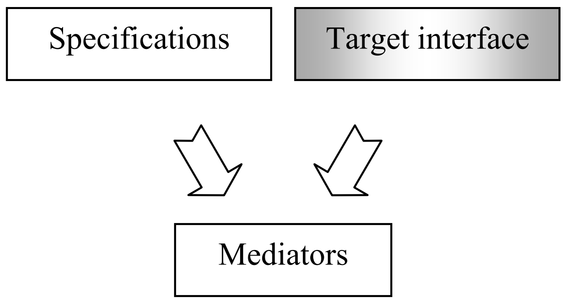
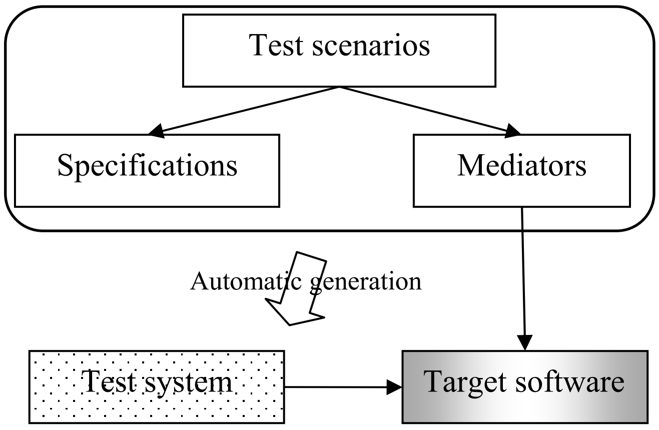
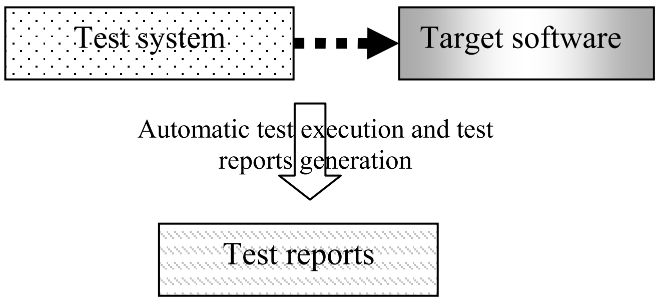
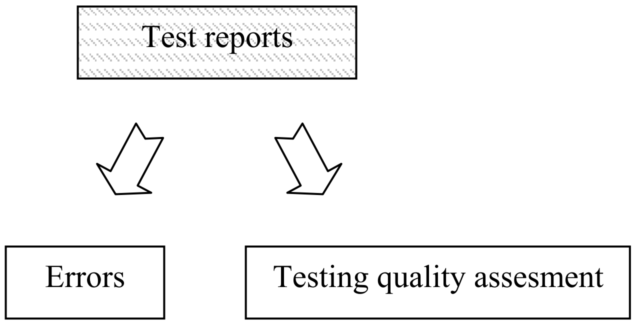
It is convenient to develop mediators in an extension of the target language, where only the transformations mentioned above should be described. Additional processing of the resulting code is required, since the mediator performs additional tasks related to specific features of the implementation environment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

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

UniTesK Test Development Process

The table below presents UniTesK process of specification development, test development and testing. The process is presented as a sequence of steps, but in practice often it becomes necessary to return to some of the previously performed steps and to revise the decisions made. So, the actual process is iterative, but its matter is a move from requirements to tests and test results analysis.

1. Analysis of functional software requirements on the base of documents available or knowledge and expertise of project participants and experts, transformation of requirements into formal specifications.	
2. Formulation of requirements to testing quality as a level of test coverage to achieve. After achieving this coverage level the testing can be finished.	 Requirements to testing quality Specifications Target coverage level
3. Development of test scenarios set that ensures the achievement of the target coverage level. Scenarios are developed on the base of specifications and do not related with some specific implementation of target system or its specific version.	

4. Binding specifications with the target software interface. The set of mediators should be developed for that.	
5. Test system construction. To obtain a ready-for-use test system specifications, mediators, and test scenarios should be translated into target programming language and then compiled into binaries.	 Test scenarios Specifications Mediators Automatic generation Test system Target software
6. Test execution with debugging of specifications, mediators, scenarios.	 Test system Target software Automatic test execution and test reports generation Test reports
7. Test results analysis. The testing quality should be assessed on the base of coverage criteria stated before. After that we can return to test scenario development to achieve higher coverage, or can finish testing and develop a report describing errors detected.	

The scheme on Figure 2 represents the process of construction of main components of UniTesK test suite architecture.

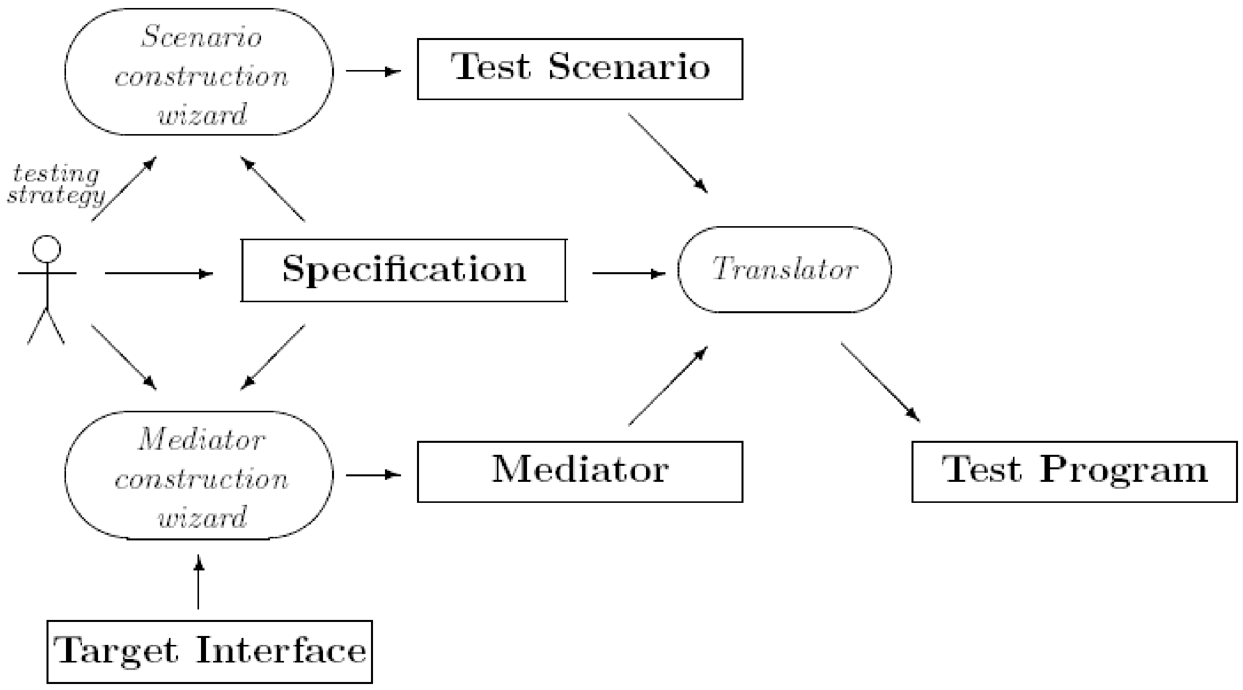


Fig. 2. Construction of UniTesK test components.

Let now consider the activities performed in course of test development in more details.

Representation of Functional Requirements

UniTesK supports an automatic generation of test oracles from contract specifications. With The target system is modeled as a set of components having interface operations with certain parameters. The environment 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 in general case depends on the internal state and operation invocation can change it.

Every operation is described by its *precondition* and *postcondition*. A precondition defines 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 ands all its parameter arrays. A postcondition imposes restrictions on the possible combination of the starting state, parameters, operation results, and the resulting state; these restrictions must be satisfied if the precondition holds when the operation is 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 their data is described along with integrity constraints on it 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 adequately describe software in various problem domains. Contract specifications can be made abstract or very detailed as required. Very important is that 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 simultaneous occurrence of the same errors in the specification and implementation.

Contract specifications are not the only 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.

Axiomatic specifications often cannot be directly transformed into general oracles, which verify system behavior correctness in response to an arbitrary input action. Therefore, axiomatic specifications are used only as additional verification criteria for certain test sequences. In such a role they can serve a base for test scenario construction.

Test Coverage Criteria Based on Specifications

The structure of contracts is used in UniTesK to define test coverage criteria, which are necessary to evaluate the quality of testing from the viewpoint of the requirements for the target software. To make possible automatic extraction of such criteria additional operators for defining *functional branches* are introduced. These statements are placed by the user in postcondition. A functional branch corresponds to a subdomain of the “same” behavior in the operation domain. 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 expressions in the postcondition for all points of the subdomain.

The control flow graph of a postcondition computation should have exactly one functional branch statement at every path from the entry to any exit. The part of the path to such a statement must not contain branching points depending on the results of the operation. Then, every admissible call of the operation can be unambiguously associated with some functional branch; therefore, the quality of testing the operation can be measured as the percentage of functional branches covered by the test. Moreover, the functional 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 possible to construct a filter that discards parameters arrays that do not enhance the coverage.

Functional branches in postconditions can help to extract automatically more detailed coverage criteria based on the structure of branching 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 branching points. It is an analog of the MC/DC criterion [MCDC] for the source code coverage.

Some disjuncts can be unattainable due to implicit semantic relations 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 semantic relations between prime logical formulas in the form of tautologies, i.e., logical expressions constructed from prime formulas that are identically true due to dependences between 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-based coverage criteria by hand as a set of predicates that depend on the parameters of operations and the state. These criteria can also be used to determine testing goals.

Test Sequence Construction

UniTesK uses FSM models of the target software in the form of test scenarios to generate sequences of test inputs on-the-fly. A scenario determines what is considered as the FSM state and which operations with what parameters arrays should be invoked in each state. In the course of a test execution, the test engine constructs an "exhaustive" path through the FSM'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 used.

When the scenario is developed, a specification-based coverage criterion can be used to determine what coverage level we want to achieve, and a set of states and transitions can be determined in such a way that the transition tour in the resulting FSM guarantees that the desired coverage level is achieved. For this purpose, it is sufficient to consider a set of predicates that determines elements of the coverage for an 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.

After that all possible intersections of these projections for all operations are constructed. So, we get a collection of sets of states such that any operation invoked for two states from the same set covers the same elements according to 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 FSM. Stimuli in this FSM 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 [FACTOR]).

In the course of testing the automatically generated filters that discard the sets of arguments that do not enhance the coverage are used. The availability of such filters makes possible to save user's efforts in computation of the parameters values needed to achieve the desired coverage; instead, one can specify a large set of parameters that surely contains all necessary values. Thus, UniTesK makes possible efficient testing intended to achieve high coverage level.

A test scenario represents an FSM 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 computation of 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 FSM models is uncommon, it makes 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 base 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, we can abstract from certain details in the specifications, thus decreasing the number of states in the resulting model (see [FACTOR]). Thus, the test generation can vary independently of specifications and, therefore, independently of the mechanism used to verify the behavior under a single input action.

Test scenarios can be developed by hands, but often they can be 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 to construct both scenarios that do not use filtration of test inputs and scenarios targeted 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. A scenario that inherits the other one can override the state computation procedure and override or enhance the set of test input actions that can be applied to the system at every state.

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 responses, we can specify *asynchronous responses* of the system representing them in the form of an operation without parameters having preconditions and postconditions.

To describe system response on concurrent inputs *the plain concurrency axiom* is used. This axiom states that the result of concurrent execution of any set of operations coincides with the execution of the same set of operations in some order. It allows to specify system behavior only in response for single input actions and to deduce its behavior when these actions are applied concurrently. Not all the software systems satisfy this axiom, but the most part of practically significant systems can be modeled with the help of models satisfying it.

Automaton models used for testing such systems are a generalization of input/output automata [IOSMA]. 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 occurrence of asynchronous response and the corresponding postcondition is satisfied after each operation or response occurrence, 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 detected.

In addition to the capabilities mentioned above, UniTesK test scenarios make 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 test scenarios that refine the target system use cases.

Another method of test scenario construction 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 in all states where it is admissible. Algebraic axioms, which require equivalence of the results 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.

Binding Specifications and Implementation

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

- ♦ The correspondence between the requirements (represented by specifications) and test suites can be controlled automatically.
- ♦ The co-verification development is supported, that is simultaneous development of the software and tests for it. This kind of development 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 behavior of the components in more situations.

Mediators can be developed by hands and thus determine nontrivial transformations between the model interface and the implementation interface. In simple cases *the mediator generation template* can be used for automatic generation of mediators. In this case the specification and implementation components should be specified and a correspondence between their operations must be established. For each operation we should specify a transformation of model parameters into the implementation ones and a transformation of the implementation results into the model ones if these transformations are not identical.

UniTesK makes 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 base 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 mediator operation, which is automatically called by the test system after every call of the target operation (if there are no concurrent calls of the target system and asynchronous responses).

If available information is insufficient for the construction of the model state (or when concurrent calls are tested or when the system can provide asynchronous responses), *the hidden state testing* is used. In this case, the model state obtained after the operation call is constructed on the base of the preceding model state, 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 contain, for each model operation, a construction of the model state obtained after the application of this operation.

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 interface 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 construction of mediators requires huge effort of highly qualified experts who know both languages very well. The required training is very costly and takes a long time.

In order to make the technology available to ordinary developers, UniTesK supports development of specifications and scenarios in extensions of widely used programming languages. To this end, a uniform system of basic concepts has been developed that are used for developing specifications and scenarios; these are the concepts of precondition, postcondition, invariant, functional branch, 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 a programming language by a corresponding construct are formulated. For languages that already have constructs 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 be easily taught to write specifications in the extension of the target language. The problem of insufficient expressiveness 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 used can be solved in several ways. First, one can prohibit using in specifications language 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.

Below you can find a table presenting examples of specifications of square root function. Specifications in extensions of different programming languages are shown in the left column of the table. The right column contains the notes on the structure of specifications.

Java	
specification **package** example;	Package declaration
class SqrtSpecification	Class declaration
{	
specification **static** **double** sqrt (**double** x)	Operation specification
reads x, epsilon	Read/write access description
{	
pre { **return** x >= 0; }	Precondition
post	Postcondition
{	
if(x == 0)	
{	
branch "Zero argument";	Functional branch definition
return sqrt == 0;	Constraints on the result
}	
else	
{	
branch "Positive argument";	Functional branch definition
return sqrt >= 0 &&	Constraints on the result
Math.abs((sqrt*sqrt-x)/x)<epsilon;	
}	
}	
}	
}	

C#	
namespace Examples	Namespace declaration
{	
specification **class** SqrtSpecification	Class declaration
{	
specification **static double** Sqrt (**double** x)	Operation specification
reads x, epsilon	Read/write access description
{	
pre { **return** x >= 0; }	Precondition
post	Postcondition
{	
if(x == 0)	
{	
branch ZERO ("Zero argument");	Functional branch definition
return **$this**.Result == 0;	Constraints on the result
}	
else	
{	
branch POS ("Positive argument");	Functional branch definition
return **$this**.Result >= 0 &&	Constraints on the result
Math.Abs((**$this**.Result * **$this**.Result - x)/x)	
< epsilon;	
}	
}	
}	
}	
}	

C	
specification **double** SQRT (**double** x)	Operation specification
reads (**double**)x, epsilon	Read/write access description
{	
pre { **return** x >= 0.; }	Precondition
coverage BRANCHES	Test coverage definition
{	
if(x == 0)	
return(ZERO, "Zero argument");	Functional branch definition
else	
return(POS, "Positive argument");	Functional branch definition
}	
post	Postcondition
{	
if(**coverage**(ZP, ZERO))	
return SQRT == 0.;	Constraints on the result
else	
return SQRT >= 0. &&	Constraints on the result
abs((SQRT*SQRT - x)/x) < epsilon;	
}	
}	
}	

Test Execution and Test Results Analysis

UniTesK tools support the automatic execution of tests developed and the automatic collection of tracing information. After a test is 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 test coverage achieved for all coverage criteria defined for a certain specification operation, and information about detected failures, which may occur 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 failure, 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 testing quality, and often helps to localize errors.

Below the examples of test reports generated by J@T tool supporting UniTesK testing of Java software are presented.

J@T Failure Report
generated: 12.12.2003 17:48:48

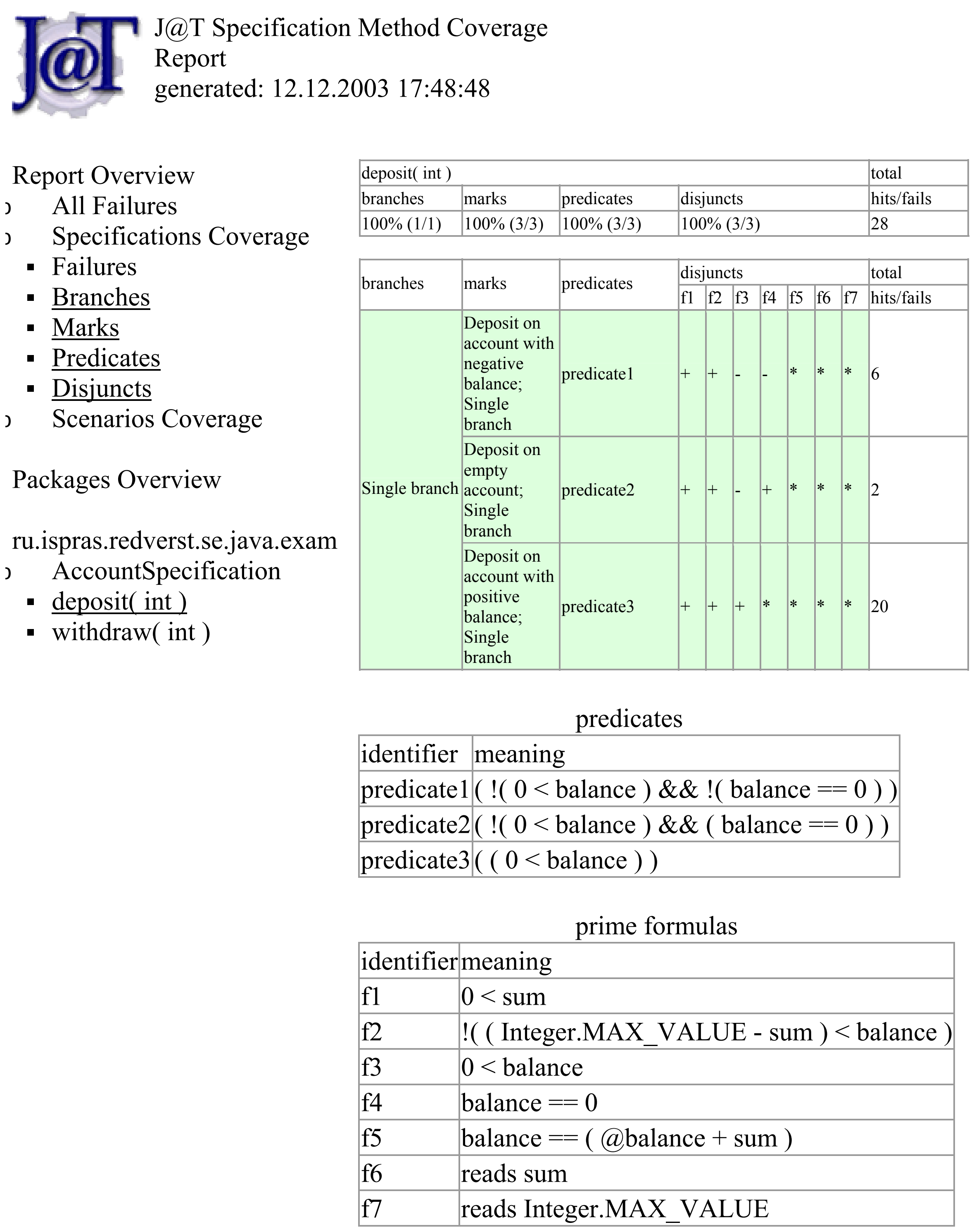
Report Overview

All Failures
- Specifications Coverage
- Scenarios Coverage

test situation

branch	Withdrawn sum is too large	
mark	Withdrawal from empty account; Withdrawn sum is too large	
predicate	(!(0 < balance) && (balance == 0)) && (((balance - sum) < -maximumCredit))	
disjunct	true	0 < sum
	false	0 < balance
	true	balance == 0
	true	(balance - sum) < -maximumCredit
	true	balance == @balance
	false	withdraw == 0
	-	balance == (@balance - sum)
	-	withdraw == sum
	-	reads sum
	-	reads maximumCredit

Fig. 3. Description of detected failure.



J@T Specification Method Coverage Report
generated: 12.12.2003 17:48:48

Report Overview
- All Failures
- Specifications Coverage
  - Failures
  - Branches
  - Marks
  - Predicates
  - Disjuncts
- Scenarios Coverage

Packages Overview

ru.ispras.redverst.se.java.exam
- AccountSpecification
  - deposit(int)
  - withdraw(int)

deposit(int)				total
branches	marks	predicates	disjuncts	hits/fails
100% (1/1)	100% (3/3)	100% (3/3)	100% (3/3)	28

branches	marks	predicates	disjuncts							total
			f1	f2	f3	f4	f5	f6	f7	hits/fails
Single branch	Deposit on account with negative balance; Single branch	predicate1	+	+	-	-	*	*	*	6
	Deposit on empty account; Single branch	predicate2	+	+	-	+	*	*	*	2
	Deposit on account with positive balance; Single branch	predicate3	+	+	+	*	*	*	*	20

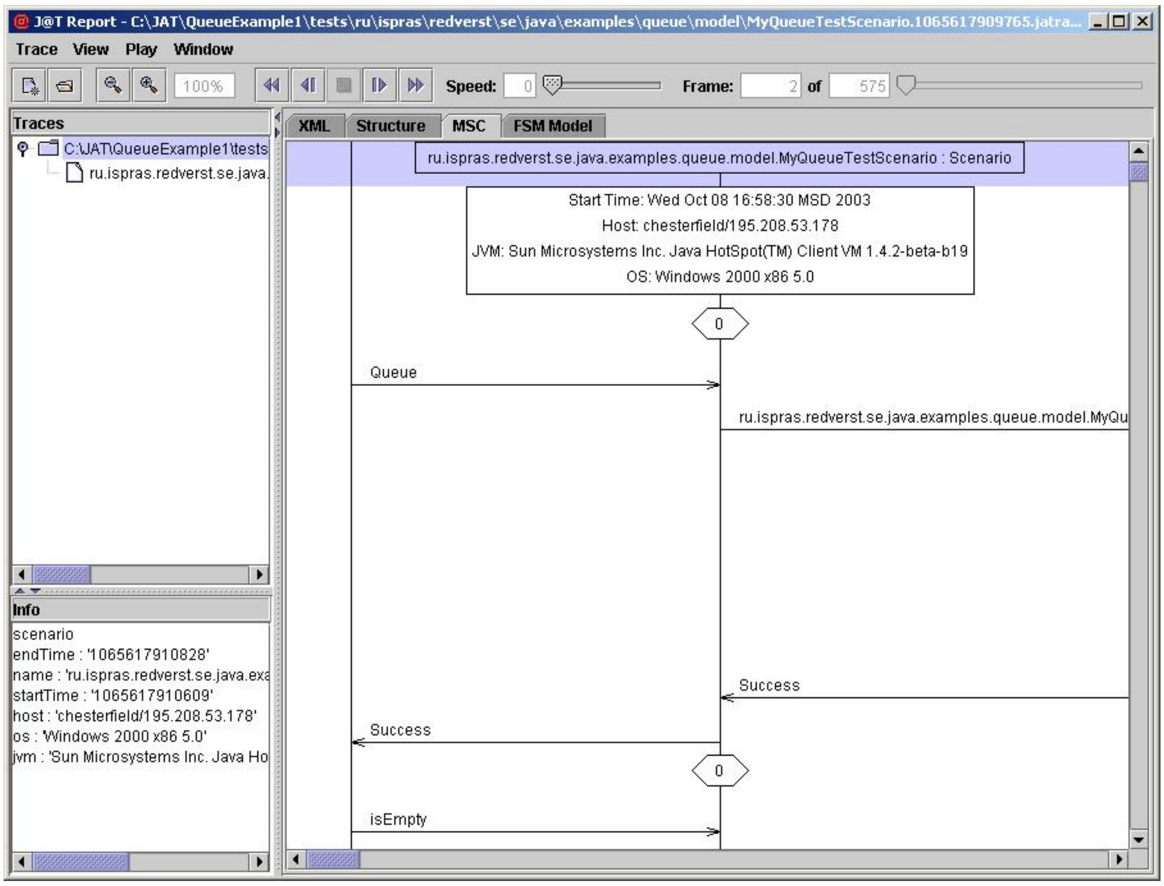
predicates

identifier	meaning
predicate1	(!(0 < balance) && !(balance == 0))
predicate2	(!(0 < balance) && (balance == 0))
predicate3	((0 < balance))

prime formulas

identifier	meaning
f1	0 < sum
f2	!((Integer.MAX_VALUE - sum) < balance)
f3	0 < balance
f4	balance == 0
f5	balance == (@balance + sum)
f6	reads sum
f7	reads Integer.MAX_VALUE

Fig. 4. Report on test coverage achieved.


J@T Report - C:\JAT\QueueExample1\tests\ru\ispras\redverst\se\java\examples\queue\model\MyQueueTestScenario.1065617909765.jatra...
Trace
View
Play
Window
100%
Speed:
0
Frame:
2
of
575
Traces
XML
Structure
MSC
FSM Model
Info
0
Queue
Success
Success
0
isEmpty

C:\JAT\QueueExample1\tests

ru.ispras.redverst.se.java.

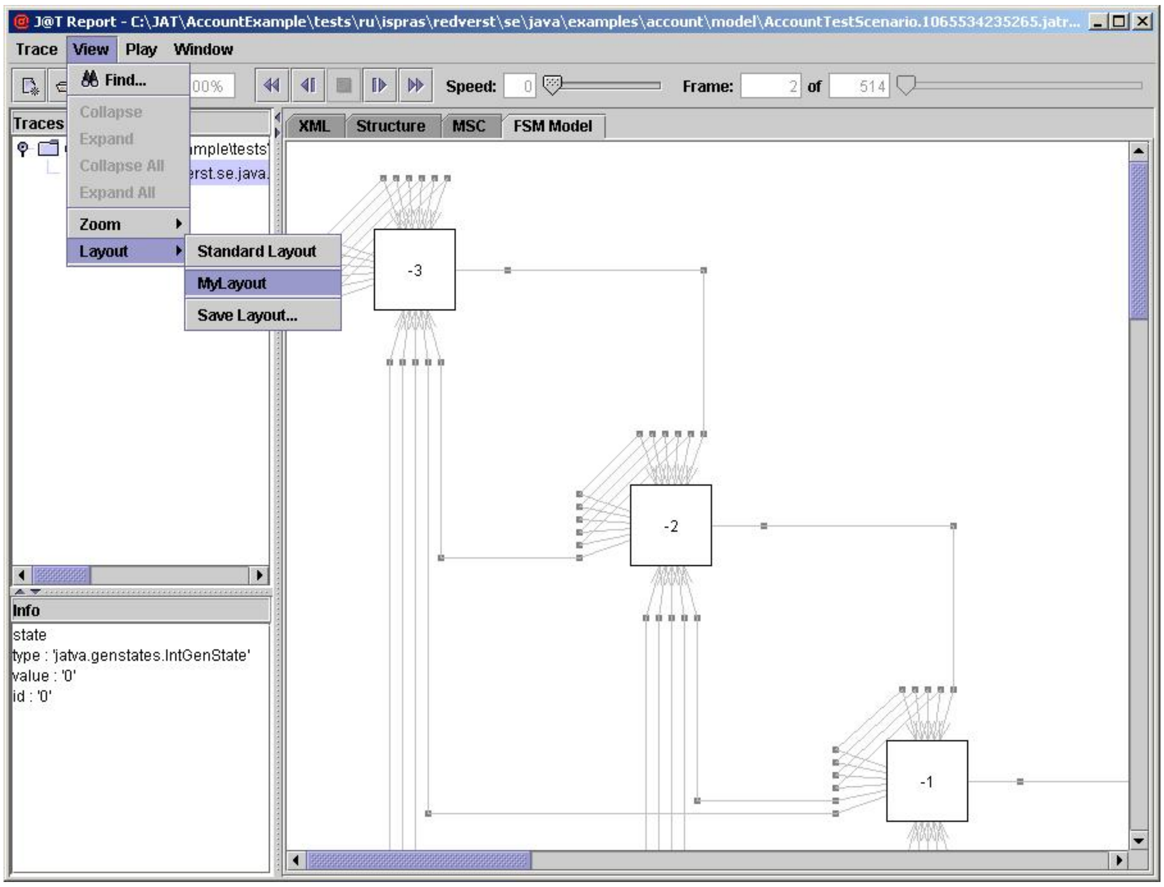
ru.ispras.redverst.se.java.examples.queue.model.MyQueueTestScenario : Scenario

Start Time: Wed Oct 08 16:58:30 MSD 2003
Host: chesterfield/195.208.53.178
JVM: Sun Microsystems Inc. Java HotSpot(TM) Client VM 1.4.2-beta-b19
OS: Windows 2000 x86 5.0

ru.ispras.redverst.se.java.examples.queue.model.MyQu

```
scenario
endTime : '1065617910828'
name : 'ru.ispras.redverst.se.java.exa
startTime : '1065617910609'
host : 'chesterfield/195.208.53.178'
os : 'Windows 2000 x86 5.0'
jvm : 'Sun Microsystems Inc. Java Ho
```

Fig. 5. Test trace in MSC representation.



```
state
type : 'jatva.genstates.IntGenState'
value : '0'
id : '0'
```

Fig. 6. Test trace represented as FSM state transition graph.

Comparison with Other Approaches to Model Based Test Development

Although almost every UniTesK feature can be found in other technologies and test development systems (sometimes, in a more advanced form), none of the approaches available in the academy 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 intended to be used in software industry. Thus, many interesting techniques developed in academic community are left beyond the scope of this survey.

The available approaches to test development mainly rely on the conventional test architecture that was 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 verifying the behavior of 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 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 it usually requires the execution of the entire set of tests.

Automatic generation of test oracles differentiates UniTesK from such tools as JUnit [JUNIT], 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 [iContract,iContractW], JMSAssert [JMS], JML [JML,JMLW], jContractor [jContr,jContrW], Jass [Jass,JassW], Handshake [Handshake], JISL [JISL] 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 [CCNET] и [ORA]).
- ♦ SLIC [SLIC] makes possible to write contract specifications in an extension of C with the use of temporal logic predicates.
- ♦ Test RealTime [TRT] developed 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 [PARASOFT] by Parasoft and Korat [KORAT] make it possible to write preconditions, postconditions, and invariants in the form of special comments in Java programs.
- ♦ ATG-Rover [TRover] uses specifications in the form of contracts written as comments in C, Java, or Verilog, which can represent predicates of temporal logics LTL and MTL.
- ♦ The family of ADL tools [ADLt] 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 [TVEC] uses preconditions and postconditions in the form of tables in the SCR notation [SCR].

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 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 [MulSaw] in the laboratory of computer science in MIT. This tool uses contracts written in JML for generation of input data for a single method in a Java class, including the object in which this method is called. The generated data 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. In this case 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 should be further processed by hands to become actual test sequences, but the possibility of such a processing is declared.

T-VEC uses special form of specifications to extract automatically information on boundary values of the regions in which a function described by those specifications behaves "identically" (cf. the definitions of functional branches in UniTesK). Test inputs are generated to cover the boundary points of functional branches of 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 targeted at high coverage in terms of criteria extracted from the internal structure of contract specifications. The majority of tools available 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 FSMs, communicating FSMs, 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 models listed above (such as SDL [SDL,ITUSDL,ITUSDLN], LOTOS [LOTOS], Estelle [Estelle], ESTEREL [ESTEREL,ESTERELL], or Lustre [Lustre]) 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 some analogs of test scenarios usually called *test purpose*s, which are 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 FSM (see for example [TPA,TPB,CADPO,TGV,TorX]). Other tools use explicitly described automaton models for the generation of test sequences targeted to achieve a certain level of coverage with respect to a certain criterion [SDLt,EST]. As has been mentioned above, UniTesK supports generation of test sequences from user-defined scenarios and an automaton model of the system; thus combining both approaches.

In terms of capabilities, GOTCHA-TCBeans [UMBTG,GOTCHA] (one of the test generation instruments of the AGEDIS project [AGEDIS,AGEDISW]) and AsmL Test Tool [ASMT,ASMTW] 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 language [Murphi], while AsmL Test Tool uses a description of the target system in the form of an abstract state machine (ASM) specification [ASMI,ASMB].

All three approaches use models of different types for constructing tests, which makes possible to generate more efficient, flexible, and scalable tests; moreover, many test 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 the test sequence.

In those tools, certain techniques for decreasing the size of model are used, which are similar to automata factorization used in UniTesK. GOTCHA-TCBeans uses a particular case of factorization that ignores values of certain fields in the model state [PROJ]. AsmL Test Tool can construct test sequences on the basis of a FSM whose states are obtained by reduction of the complete state of the ASM to the set of values of elementary logical formulas used in the description of ASM transitions [FfASM].

The main differences between UniTesK from one side and GOTCHA-TCBeans and AsmL Test Tool from the other side are the support of programming language extensions for specification development, the use of contract specifications, automatic control of specification coverage, and the use of filters targeted to produce test inputs that enhance coverage.

Experience of UniTesK Applications

The development of tools supporting UniTesK technology was started at the end of 1999. By that time several joint projects with Nortel Networks had been conducted where KVEST technology had been used for test development and regression testing of complex telecommunication software systems. Examples of the systems, to which KVEST was applied, are the following.

- The kernel of switch real-time operating system (about 250K lines of code written in Protel language similar to C).
- The message management system.
- The framework of base level telecommunication services.
- The distributed peer-to-peer messaging system.

The first UniTesK implementation ready for real use was CTesK intended for testing C programs. The simplified version of CTesK tools was developed by the end of 2000. By the middle of 2001 they were actively used in the research project supported by Microsoft Research. The project goal was to test the implementation of new generation Internet protocol IPv6. Besides, it was necessary to check the applicability of UniTesK to telecommunication protocol implementation testing. By the end of 2001 the project was finished. The serious errors were found that can cause crash of operating system on any node in IPv6 network. The suitability of UniTesK for such kind of problems was approved by the fact that none of a dozen groups working worldwide on testing the same implementation managed to find these errors. The novel approach to testing distributed and concurrent systems based on asynchronous reaction specifications and IOSM models was successfully applied in this project.

In 2003 the pilot project using CTesK to test real-time software was conducted (algorithms of navigation control, GosNIIAS). In spite of short project terms the serious error was found.

In the middle of 2002 the first commercial version of J@T tool intended for testing Java software war developed[1]. By the end of 2003 several pilot projects using J@T were conducted, the following systems were tested in these projects:

- The banking client data management system implemented in three-tier architecture on the base of EJB technology (Luxoft).
- The query creation and query data transformation subsystem in distributed banking Web application (Tarang)
- The DBMS communication subsystem in the framework for multi-tier business application development (VisualSoft)
- The client working time control subsystem in billing system (VebTel).

All the projects mentioned above demonstrated the high effectiveness of UniTesK technology. Errors were found in all the systems tested, some of them are recognized by customers as very serious. For example, the situation was found when the results of unprocessed transactions were

[1] J@T tool can be also used to test C++ software through additional mediator level.

saved in database. Some of the projects mentioned were conducted in very short terms — about 1-2 days.

In 2001 RedVerst group and Intel started the joint research project on adaptation of model based testing techniques to optimizing modules of compilers. At first the experiments were conducted on free compilers such as gcc and Open64. Even the first case studies showed that high test coverage can be achieved with the help of techniques used. For optimizing modules of gcc compiler the tests constructed covered about 90-95% of operators, which is very high for industrial testing. Then, the experiments were conducted on the last generation of Intel compilers for Itanium 64-bit platform. The effectiveness of the adapted technology was approved there too.

By the middle of 2003 on the base of the experience gained the toolkit for model based test construction for compilers and other text processors was developed. The toolkit was then applied to generation of protocol data for protocol implementation testing.

The adaptation of UniTesK to compiler testing required to solve several serious problems and to develop new techniques of test construction. The following tasks were solved.

- The description of model language, which is simplification of the compiler's input language.
- The development of iterators of model language constructs.
- The test oracle construction to check the correctness of optimization.

The toolkit for compiler testing was included in the UniTesK tools family. It demonstrated that model based test construction techniques are rather general and can be applied in wide domain.

By the end of 2003 the first version of Ch@se tool intended for .Net application testing was developed. It is based on the extension of C# language and provides one more implementation of UniTesK technology.

The full list of projects where UniTesK was applied and description of their results can be found on RedVerst web site [RedVerst].

Conclusion. Open Problems and Lines of Future Development

The experience of UniTesK applications along with experience of other groups working on model based testing methods shows that this approach is quite effective. What are the obstacles that prevent wide use of model based testing techniques? They can be divided into three groups.

- Methodical problems concerned with methods of specification and test development.
- Technical problems concerned with unification of specification development methods and tools supporting such a development.
- Organizational problems concerned with transition of the methods and tools to the industrial software development processes.

The complexity of the problems increases from the first group to the third one, which is shown by the experience of many groups using model based testing methods to test industrial software. Almost in all cases the modeling method can be found that help to achieve both high quality of testing and affordable cost of test development. So, lack of the appropriate theoretic and methodical developments is not the main obstacle for use of the approach discussed.

Now there is no general consent on the methods and notations used for model based testing, and there is no generally recognized uniform verification suite architecture[2], tools for static or dynamic analysis of software and representations of results of such analysis are far from unification and interchangeability. Nevertheless, the need for such unification already exists. The widely known attempt to unify test suite architectures is UML Testing Profile [UML-TP] developed in the framework of MDA (Model Driven Architecture, [MDA]) program. The authors of UML Testing Profile maybe have tried to find a compromise suitable for all the members of OMG consortium

[2]Verification suite is a collection of componets of models and tests constructed on their base considered with all the relations between them.

and suggest the architecture targeted mostly for manual test development. This approach can be used in test development based on use case specifications, but it becomes a limiting factor if we use automata models or contract specifications for test development. Maybe this is caused by the lack of understanding of advantages of contract specifications, and OCL (Object Constraint Language) in particular, for industrial software development. Thus, the problem of unification is not only technical. It is important to find a unified approach that will enable test development on the base of any model kinds using any technology (or manually) in the single framework.

Organizational problems are the most critical ones today [Robinson, Manage]. As we already mentioned above, the successful test development technology should be integrated in the conventional development processes and should not require long and expensive staff training. Are there any ways to combine mode based testing techniques with conventional development methods without breaking the latter?

It seems that there are no ways for absolutely painless introduction of model based methods in the industry, because of problems of staff preparation, not only technical one, but also managers. Now test development and testing are regarded mostly as only auxiliary activities in the industry. Testers are often considered as low-skilled staff, without requirements to have basic programming skills or to know some programming languages. In attempt to simplify the collaboration between testers and developers project managers often target testers only on system testing, without development of module or integration tests. In this case an illusion appears that testers have enough time to prepare quality tests. But in reality by the evaluation testing time the system can demonstrate reliable operation only on mostly used scenarios and requires a lot of additional work for most part of uncommon use cases.

It is quite obvious that the negative tendencies mentioned cannot be “broken” by force, they can only become outdates in the course of development processes evolution. Now such an evolution can be supported by assigning additional verification and testing activities to designers and developers. These activities will become conventional only when they lead to obvious positive changes in the main activities of software designers and developers. This is the reason to integrate tools for test development automation in the development environments, making them accessible not only for testers, but also for developers.[3]

To extend the model based testing community we need to develop and apply new forms of education to make it more effective for these methods. All the UniTesK tools are supplemented with user documentation and training materials. Two forms of mastering the tools are suggested — traditional university course and intensive training course. University course requires 15-30 hours and training requires spending 8 hours in a day and 3-4 days. University course materials are provided with academic licenses for UniTesK tools for free. They are also supplemented with examples of specifications and tests for various applications.

The main strategic goal of further UniTesK development is to develop full scale industrially applicable technologies for testing any kind of industrial software.

For successful progress of model based testing techniques the following three aspects should be developed simultaneously. First, the functionality of these techniques should be extended, second, the integration between them and conventional development processes should be made as smooth as possible, and third, the usability of the techniques should be increased. The most important task is to increase the usability of model based testing for the most common development tasks.

The extension of the model based testing domain should be provided in the following directions.

1. Development of full scale technologies for testing all components of compilers, interpreters, DBMS, and other applications processing requests in formal languages.

[3]It is a common knowledge that the developer cannot test the system he develops. But when the models used are different from implementation in their structure such testing become effective.

2. Development of automated model based testing technology for graphical user interfaces (GUI).
3. Development of full scale testing technologies for distributed systems, especially for multi-tier and client-server applications widely used now.
4. Development of testing techniques for components with very complex functionality and narrow interface as task schedulers, garbage collectors, transaction managers, etc.
5. Development of testing techniques for applications with elements of artificial intelligence such as image recognition applications, intellectual agents, applications on the base of fuzzy logic, etc.

Besides providing solutions for problems of testing itself, we need software quality metrics that can demonstrate advantages of model based testing methods. Nowadays, most part of metrics used for evaluation of resulting software quality or to control the state of development process is suited for manual test development. But when they are applied to the projects using automated model based test generation, they show a paradoxical picture. To demonstrate advantages of model based testing we need methods to clarify the relation between the original informal requirements, their models used, and resulting tests. Another problem is lack of suitable reusability metrics that can evaluate impartially impact of changes in requirements, in testing quality requirements, in the target software architecture, in the development technologies used on the tests. Development and application of such metrics in the industry can increase the importance of technologies intended to provide high quality software, that in turn will inspire new developments in testing automation and model based testing.

References

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

[AGEDIS] I. Gronau, A. Hartman, A. Kirshin, K. Nagin, and S. Olvovsky. A Methodology and Architecture for Automated Software Testing. Available at http://www.haifa.il.ibm.com/projects/verification/gtcb/papers/gtcbmanda.pdf

[AGEDISW] http://www.agedis.de/

[ASMB] E. Börger and R. Stark. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.

[ASMI] Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In Current Trends in Theoretical Computer Science, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, pp. 266–292.

[ASMT] 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.

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

[AsSM] I. Burdonov, A. Kossatchev, V. Kuliamin. Asynchronous automata: classification and testing methods. Proceedings of ISP RAS, 4:7-84, 2003.

[ASSUM] S. Fujiwara and G. von Bochmann. Testing Nondeterministic Finite State Machine with Fault Coverage. IFIP Transactions, Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems, 1991, Ed. by Jan Kroon, Rudolf J. Heijink, and Ed Brinksma, 1992, North-Holland, pp. 267–280.

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

[BP] 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.

[CADPO] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. INRIA Technical Report TR-254, December 2001.

[CCNET] M. Barnett and W. Schulte. Contracts, Components, and their Runtime Verification on the .NET Platform. Technical Report TR-2001-56, Microsotf Research.
[DBCA] Bertrand Meyer. Applying `Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992, pp. 40–51.
[DBCE] Bertrand Meyer. Eiffel: The Language. Prentice Hall, 1992.
[DBCO] Bertrand Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall, 1997.
[DETFSM] I. Burdonov, A. Kossatchev, V. Kuliamin. Irredundant algorithms of directed graphs traversal. Deterministic case. Programming and Computer Software, 2003.
[EST] 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.
[Estelle] ISO/TC97/SC21. Information Processing Systems — Open Systems Interconnection — Estelle — A Formal Description Technique based on an Extended State Transition Model. ISO 9074:1997, International Organization for Standardization, Geneva, Switzerland, 1997.
[ESTEREL] G. Berry. The Foundations of Esterel. In Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling, and M. Tofte, editors, MIT Press, 1998.
[ESTERELL] F. Boussinot and R. de Simone. The Esterel language. Proc. IEEE, vol. 79, pp. 1293–1304, Sept. 1991.
[FACTOR] I. Burdonov, A. Kossatchev, V. Kuliamin. Application of finite automatons for program testing. Programming and Computer Software, 26(2):61–73, 2000.
[FfASM] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In Proc. of ISSTA'2002. Also: Microsoft Research Technical Report MSR-TR-2001-97.
[FMEAD] http://www.fmeurope.org/databases/fmadb088.html
[GOTCHA] http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html
[Handshake] A. Duncan and U. Hlzle. Adding Contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, 1998.
[iContract] R. Kramer. iContract — The Java Design by Contract Tool. In Proceedings of TOOLS26: Technology of Object-Oriented Languages and Systems, pp. 295–307. IEEE Computer Society, 1998.
[iContractW] http://www.reliable-systems.com/
[IOSMA] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand. Towards Analysing and Synthesizing Protocols. IEEE Transactions on Communications, COM-28(4):651–660, April 1980.
[ITUSDL] ITU-T, Recommendation Z.100: Specification and Description Language (SDL), ITU-T, Geneva, 1996.
[ITUSDLN] ITU-T, Recommendation Z.100 Annex F1: SDL formal definition — General, 2000.
[Jass] D. Bartetzko, C. Fisher, M. Moller, and H. Wehrheim. Jass — Java with assertions. In K. Havelund and G. Rosu, editors, Proceeding of the First Workshop on Runtime Verification RV'01, Vol. 55 of Electronic Notes in Theoretical Compter Science, Elsevier Science, July 2001.
[JassW] http://semantik.informatik.uni-oldenburg.de/~jass
[Jatva] Igor B. Bourdonov, Alexey V. Demakov, Andrew A. Jarov, Alexander S. Kossatchev, Victor V. Kuliamin, Alexander K. Petrenko, Sergey V. Zelenov. Java Specification Extension for Automated Test Development. Proceedings of PSI'01. LNCS 2244, pp. 301–307. Springer-Verlag, 2001.
[jContr] M. Karaorman, U. Holzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. Technical Report TRCCS98-31, University of California, Santa Barbara. Computer Science, January 19, 1999.
[jContrW] http://jcontractor.sourceforge.net/

[JISL] P. Muller, J. Meyer, and A. Poetzsch-Heffter. Making executable interface specifications more expressive. In C. H. Cap, editor, JIT'99 Java-Informations-Tage 1999, Informatik Aktuell. Springer-Verlag, 1999.
[JML] A. Bhorkar. A Run-time Assertion Checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 2000.
[JMLW] http://www.cs.iastate.edu/~leavens/JML.html
[JMS] http://www.mmsindia.com/JMSAssert.html
[JUNIT] http://www.junit.org/index.htm
[KORAT] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. Proc. of ISSTA 2002, Rome, Italy. Jul 2002.
[KVEST] 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.
[LOTOS] ISO/IEC. Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807:1989, International Organization for Standardization, Geneva, Switzerland, 1989.
[Lustre] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. Proc. IEEE, vol. 79, pp. 1305–1320, Sept. 1991.
[LY] 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.
[MCDC] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, pp. 193–200, September 1994.
[MSRIPreport] http://www.ispras.ru/~RedVerst/RedVerst/White Papers/MSRIPv6 Verification Project/Main.html
[MulSaw] http://mulsaw.lcs.mit.edu/
[Murphi] http://verify.stanford.edu/dill/murphi.html
[NDFSM] I. Burdonov, A. Kossatchev, V. Kuliamin. Irredundant algorithms of directed graphs traversal. Nondeterministic case. Programming and Computer Software, 1, 2004.
[Manage] D. Stidolph, J. Whitehead. Managerial Issues for the Consideration and Use of Formal Methods, LNCS 2805, 2003, pp.170-186.]
[MDA] http://www.omg.org/mda/
[OPT] A. Kossatchev, A. Petrenko, S. Zelenov, and S. Zelenova. Using Model-Based Approach for Automated Testing of Optimizing Compilers. In Proccedings of Intl. Workshop on Program Undestanding, Gorno-Altaisk, 2003.
[ORA] L. Baresi and M. Young. Test Oracles. Tech. Report CIS-TR-01-02. Available at http://www.cs.uoregon.edu/~michal/pubs/oracles.html
[PARASOFT] http://www.parasoft.com
[Parnas] D. Peters and D. Parnas. Using Test Oracles Generated from Program Documentation. IEEE Transactions on Software Engineering, 24(3):161–173, 1998.
[PROJ] G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected state machine coverage for software testing. Proc. of ISSTA 2002, Rome, Italy. Jul 2002.
[RedVerst] http://www.ispras.ru/groups/rv/rv.html
[Robinson] H. Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. In proceedings of 1-st ECMDSE, 2003
[SCR] C. Heitmeyer. Software Cost Reduction. Encyclopedia of Software Engineering, Two Volumes, John J. Marciniak, editor, ISBN: 0-471-02895-9, January 2002.
[SDL] J. Ellsberger, D. Hogrefe, and A. Sarma, SDL — Formal Object-Oriented Language for Communicating Systems, Prentice Hall, 1997.
[SDLt] 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).
[SLIC] T. Ball and S. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report, MSR-TR-2001-21, Microsoft Research, January 2002.

[TGV] 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.
[TorX] 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.
[TPA] J. Grabowski, D. Hogrefe, R. Nahm. Test case generation with test purpose specification by MSCs. In O. Faergemand and A. Sarma, editors, 6th SDL Forum, pages 253–266, Darmstadt, Germany, North-Holland 1993.
[TPB] 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.
[TRover] http://www.time-rover.com
[TRT] http://www.rational.com
[TVEC] http://www.t-vec.com
[UMBTG] 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.
[UML-TP] UML Testing Profile. http://www.omg.org/docs/ptc/03-07-01.pdf
[UniArch] I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. Proc. of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.
[UNITESK] http://unitesk.ispras.ru