UniTesK Test Suite Architecture

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

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

Abstract. The article presents the main components of the test suite architecture underlying UniTesK test development technology, an automated specification based test development technology for use in industrial testing of general-purpose software. The architecture presented contains such elements as automatically generated oracles, components to monitor formally defined test coverage criteria, and test scenario specifications for test sequence generation with the help of an automata based testing mechanism. This work stems from the ISP RAS results of academic research and 7-years experience in industrial application of formal testing techniques [1].

Keywords: specification based testing, partition testing, automata based testing, test suite architecture.

1 Introduction

An automated technology of industrial test development applicable for general purpose software is a dream of several generations of test designers and testers. During past decades many views concerning both the possibility of such a technology and various approaches to its implementation were expressed in research community. At the same time, the industry has attained such a level of software complexity that some rigorous testing technology is recognized as an urgent need.

But the industry is not in a hurry to accept the first technology to appear. Industrial software development processes require some “critical mass” of features from test development techniques to benefit actually from them. RedVerst [1] group of ISP RAS proposes UniTesK test development technology as a candidate. UniTesK is a successor of KVEST test development technology [2, 3] developed by RedVerst group of ISP RAS for Nortel Networks. UniTesK is based on a ground experience in specification based testing obtained from 7-years work in several software verification projects performed with the help of KVEST technology. The total size of software tested in these projects is about 0.5 million lines of code. UniTesK tries to keep all the positive experience of KVEST usage. It also introduces some improvements in flexibility of technology, heightens the reusability of various artifacts of test development process, and lessens the skills required to start using the technology in an effective way.

To be successful, a test development technology should be based on considered test suite architecture. Test suite architecture should consist of clearly described components with neatly defined responsibilities, interfaces, relations, and interactions with each other, but all this is not sufficient. UniTesK technology is based on a new approach to automated test development: it proposes some architecture framework that can be used to construct effective tests for almost any kind of software system and makes testing process completely automated. The technology determines the immutable part of this framework and components depending on target system or test goals. Then, it defines the procedures that help to minimize the amount of manual work necessary to develop these components.

Although completely automated test generation is impossible, except for some very simple or very special cases, our experience shows that the work required to produce the manual components is usually much more simple than the one required to develop the similar test suite in traditional approaches. The use of the UniTesK technology is similar to the use of rich libraries of templates and classes — the developer specifies only actually important details and control specifics, uses library elements for routine tasks, and obtains the result without doing much tedious and mechanical work.

UniTesK technology can be used to produce tests for any kind of software. For each reasonable criterion of correctness of the system behavior we can construct a test, which convinces us of this correctness if it is successful, or can find some incorrectness. In simple case we can produce such a test at once, in more complex ones several iterative steps of test development is required.

This article presents the main elements of the test suite architecture underlying UniTesK test development technology. We also tried to provide the definitions of the main concepts used in UniTesK method and to supply each concept and architecture component with comments containing arguments in favor of the architecture decisions proposed. See [8] for some additional arguments.

2 Goals of Testing Determine Test Suite Structure

Since there is no sense in discussion of test suite architecture used in a technology irrespective from testing goals and main techniques used to test something in this technology, we start with some general considerations that clarify the goals of UniTesK and methods used to reach them.

The main question, the answer to which lies in the heart of each testing technology, is "What is the main goal of testing?" The short form of this answer accepted by UniTesK test development technology is that *testing should demonstrate that the system under test works correctly.* This answer raises more questions: "What is a correct work of the system under test? And how it can be demonstrated?" Let us try to give detailed answers.

The system under test works correctly if it behaves according to its requirements. This statement holds for all kinds of requirements, but UniTesK technology pays more attention on so-called *functional requirements.* They describe what the system should do regardless of methods used and such issues as scalability, performance, dependability, etc. To make requirements more clear and to enable automatic checking of system's behavior accordance to requirements, they should be formulated in a rigorous, clear, and unambiguous form. Such form is often called *formal specifications.* So, UniTesK approach is a kind of *conformance testing* — it presumes that we have formal specifications of the system behavior and they are given in such a form that enables us to generate *an oracle,* a program that can check the results of the target system's work against the constraints given in specifications.

Unfortunately for testing, most software systems are so complex and can interoperate with their environment in so many different ways, that there is a huge, often infinite, set of possible testing situations. To demonstrate the correct work of the system by trying all possible situations in such a case is obviously impossible. The only way to actually rigorous and convincing arguments in favor of correct or incorrect work of the system is to consider the structure of the system implementation and specifications and to look for some natural equivalence on the set of all possible situations induced by this structure. The main thesis of *partition testing,* also used by UniTesK technology, is that there are often a finite number of equivalence classes of all situations, which possess the following property: if the system behaves in correspondence with its specifications in one situation of such a class, then it behaves so in every situation of the same class, and, conversely, if it behaves improperly in one situation of a class, then it does the same way in every situation of the same class. Obviously, if this statement holds, we can test the behavior of the target system only in a finite number of situations representing all the equivalence classes.

The percent of situations (from some set of testing situations) that are tested during some test is called *test coverage.* The corresponding set of testing situations is called *coverage model* or *test coverage criterion.* The partition described in the previous paragraph can be used to define coverage model. For practical reasons we cannot restrict our consideration by only one coverage model. In an industrial project other coverage models are often considered as a base measure of the effectiveness of a test. So, to be useful in the industry a test development method should be able to provide tests based on different test coverage criteria for target component's domain. UniTesK technology does so. The criterion chosen as a test coverage criterion for a test we call *the target criterion* of this test.

Since we want to be able to test the component with different target criteria, we need to generate a kind of *universal oracle* from its specifications. Such an oracle should be able to check the correctness of the component's behavior for an arbitrary input (see [3–5] for more details on automatic generation of such kind of oracles). It is different from commonly implied by the term "oracle" *input/outcome oracle,* which can be used only for the prescribed input.

Let us have a more detailed look at possible mechanism of testing that is intended to achieve some coverage criterion. In most cases the only interesting part of the system's behavior is its interactions with its environment, when an environment acts on it in some way and the system reacts in some way. Other aspects, as the internal state of the system, are considered only in so far as they have an influence on possible system's reactions. So, from this point of view the system can be adequately modeled by some *automaton* having some states and transitions. Each transition is caused by some external action, or input, and, along with moving the automaton into other state, produces some reaction, or output.

The target coverage criterion defines some equivalence relation on the transitions of the modeling automaton of the system under test. We often can transform this automaton into an other one in the following way: a state of the resulting automaton corresponds to some set of states of the initial one, a transition of the resulting automaton corresponds to a set of equivalent transitions of the initial one, one for each state corresponding to the starting state of the transition in the resulting automaton, and each equivalence class of initial transitions has at least one corresponding resulting transition in some resulting state (the particular case of such transformation, factorization technique, is discussed in [6]). If we can construct a traversal of the resulting automaton (a path that contains each transition at least once) and we can find the corresponding path on the initial automaton and execute the corresponding sequence of calls, we shall achieve the target criterion. Obviously, we cover all the transitions of the resulting automaton; hence, on the initial one we cover at least one transition of every equivalence class as we need.

So, we see that the general purpose testing mechanism can be based on traversal of finite automata. This is one of the main points of UniTesK technology.

Let us now look at the details of UniTesK test suite architecture.

3 Details of UniTesK Test Suite Architecture

The core of UniTesK test suite is the traversal mechanism for finite automata. To provide additional flexibility, it is divided into two parts: *a test engine component* encapsulating a traversal algorithm of finite automata of some wide class, and *a test sequence iterator component,* which contains all the details of the particular automaton. Test engine and test sequence iterator interact via well-defined interface consisting of three following operations defined in test sequence iterator.

– `State getState().` This operation returns the identifier of the current state of the automaton. State identifiers can be stored by test engine to facilitate a traversal, but the only thing it can do with them is comparison, which shows whether two identifiers designate one state of the automaton under traversal or two different states.
– `Input next().` This operation seeks for the next input symbol in the current state, which has not been applied yet during this traversal. If there exists some, it returns anyone of such symbols, else it returns `null.` The objects of `Input` type are identifiers of input symbols. As state identifiers, they also may be stored by test engine and can be compared with each other.

– `void call(Input param)`. This operation applies the input symbol identified by the `param` object in the current state. It actually performs the corresponding transition in the automaton under traversal.

Notice, that the interface specified requires that the traversal algorithm implemented by test engine component is able to work on base of only data provided by these operations. This means, in particular, that test engine has no full description of the automaton under traversal. It can use only the structure of already traversed part of it. We call algorithms of automata traversal that require only this information *undemanding.*

Why the use of undemanding automata traversal algorithms is justified? It may seem that more traditional automata testing based on full description of the state-transition graph is more appropriate. But, notice, that undemanding traversal algorithm requires only the information on applicable input symbols in each state, and traditional approach needs also to know the end of each transition (for nondeterministic automata — all possible ends). Our experience shows that the first way to describe an automaton is much simpler and more scalable on the size of the automaton under test. To obtain the information on all the possible ends of each transition we need to examine deeply the specifications. So, along with additional data, full description of an automaton requires much more human work, because such an analysis can hardly be automated in usual case. But, if we already have the specifications, why don't describe only applicable input and let oracles generated from these specifications check the correctness automatically, instead of hard manual work?

UniTesK technology is intended to use implicit specifications that describe only properties of admissible input. So, the specifications contain enough information to generate inputs only for very simple software components. In general case, it takes much less effort to specify necessary input by hand than to derive it from such specifications (see below the discussion of the structure of test sequence iterator component).

The main idea of UniTesK testing technique is separating test sequence generation from the behavior verification. Test engine and test sequence iterator are responsible for test sequence generation and require only a part of description of an automaton under test — the set of states and applicable input symbols for each state. Oracles of target components are called during the work of test sequence iterator's `call()` method and, in turn, call the corresponding target methods and perform the verification of their behavior against the specifications.

Fig. 1 shows the basic architecture of UniTesK test suite. We do not consider here auxiliary components responsible for trace gathering and run-time test support, because they have no UniTesK specifics.

Let us say some words on the origin of the components presented. Test engine component is a predefined part of a test suite. There are several test engine components intended to traverse different classes of finite automata. The test developer does not need to write such a component himself, instead it should use one of the existing ones. Oracles are supposed to be generated automatically from specifications, which in turn are always developed by hand.

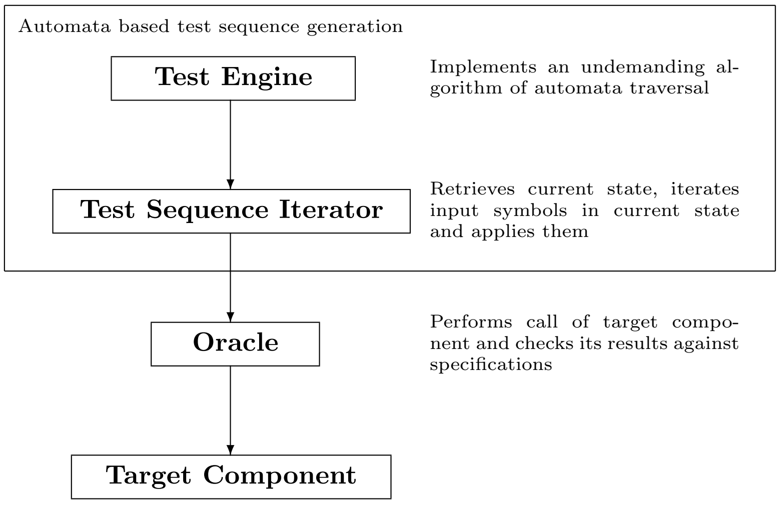
Automata based test sequence generation
Test Engine
Implements an undemanding algorithm of automata traversal
Test Sequence Iterator
Retrieves current state, iterates input symbols in current state and applies them
Oracle
Performs call of target component and checks its results against specifications
Target Component

Fig. 1. Basic architecture of UniTesK test suite

Now we consider test sequence iterator component structure in more details paying more attention to the mechanism of iteration of applicable input symbols.

Test sequence iterator should provide all possible input symbols for each state of the automaton under traversal. Remember, that we construct this automaton from the behavior of the system under test and target coverage criterion. Input symbol of the automaton under traversal corresponds to some class of possible inputs for the system under test.

An arbitrary coverage criterion can be described by a number of predicates depending on the target operation identifier, a list of its input parameters, and system's state. Each of these predicates determines one coverage task. The set of predicates, describing the criterion based on the structure of implementation or specification could be extracted from them, but it is not an easy task in general case. To facilitate this work, UniTesK technology requires specification designer to emphasize the basic functionality partition of the specified operation domain by means of special constructs, `branch` operators. The elements of this basic partition correspond to subdomains where the specified operation has substantially different functionality.

But how can we use the predicates describing coverage criterion? In most simple cases we can consider and solve the corresponding boolean equations to produce automatically the set of test cases that ensures the test coverage needed. We have seen that it is impossible in general and even for common software components used in the industry, because, for example, the equations obtained can be unsolvable by an algorithm. In spite of the fact that all the data in real computers are from some finite sets, such equations are practically unsolvable due to enormous amount of time required for that.

Thus, the general case solution should allow the test designer to facilitate test case generation with some handmade components of test system, which are called *iterators* in UniTesK. But the predicates defined above do remain useful. They can be used to filter the test cases provided by iterators, and, so, iterators need not to be very precise. An iterator used for some component under test may provide a wide range of possible inputs, including at least one representative of each coverage class. In UniTesK test case filters generated from the predicates describing the target coverage criterion are called *coverage trackers.*

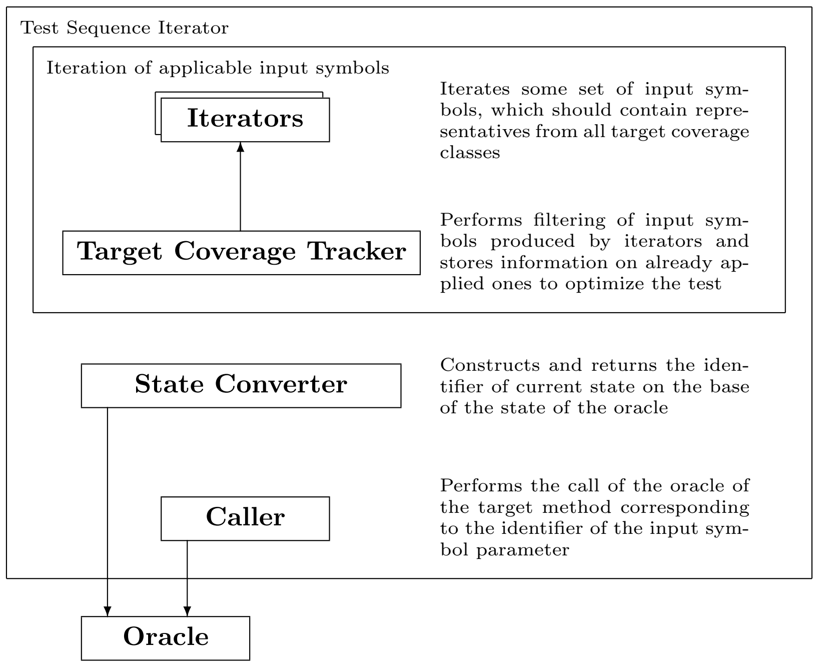
Test Sequence Iterator
Iteration of applicable input symbols
Iterators
Iterates some set of input symbols, which should contain representatives from all target coverage classes
Target Coverage Tracker
Performs filtering of input symbols produced by iterators and stores information on already applied ones to optimize the test
State Converter
Constructs and returns the identifier of current state on the base of the state of the oracle
Caller
Performs the call of the oracle of the target method corresponding to the identifier of the input symbol parameter
Oracle

Fig. 2. Typical structure of a test sequence iterator component

Such coverage tracker is used as a part of test sequence iterator component. It determines the coverage task corresponding to a call of the target operation generated by an iterator and checks whether this task has been achieved before during the test. If not, this call is considered as the next unapplied input symbol, and the tracker marks the corresponding coverage task as covered (because the test engine actually performs this call later), else the coverage tracker pushes the iterator for the next call. When all the coverage tasks are achieved, the coverage tracker reports that there are no unapplied input symbols and the test can be finished.

Other functionality of test system iterator is represented by methods `getState()` and `call()`. As our experience shows, in some cases both of them can be generated automatically from the specification of the target component. When we consider the coverage criteria based only on coverages of component's operations domains, the method `call()` can be generated in general case. For most testing tasks it is enough, but sometimes, when we need to cover some specific sequences of target operation invocations, we should write part of this method by hand. For these reasons the possibility to do so exists in the UniTesK technology.

Fig. 2 shows the structure of mechanism iterating applicable input symbols used in UniTesK.

So, as we see, test sequence iterator has a complex structure, a bulk of which is usually generated automatically — target coverage trackers and caller implementing `call()` method. Sometimes a caller needs some manual code and sometimes state converter component and iterators can also be generated. To provide a comprehensible description of all the parts of test sequence iterator UniTesK method proposes a form of *test scenario,* which can be written by hand entirely or generated and then tuned up as needed. Test scenario serves as a source for test sequence iterator generation. For more detailed description of the structure and possible syntax of test scenarios see [7, 10]. Our experience shows that the ratio of generated code size to manual code size in test sequence iterator component is usually more than 4:1.

One more important point of UniTesK test suite architecture is the use of adapter pattern (see [9] on detailed description of this pattern) to bind specification and implementation of the target component. For historical reasons we call adapters, which have specification interface and implement it on the base of the implementation under test, *mediators.*

To be able to produce reusable and repeatable tests is crucial for industrial test development technology. UniTesK supports reusability of tests and specifications by using more abstract specifications, which can be applicable to several versions of the target component. To perform a test in such a situation, we should have something to bind the immutable specification with changing implementation. One of the most simple and at the same time flexible solutions is to use mediator component having the interface of specification and implementing it with the help of implementation. Such an approach allows us to change the level of abstraction of specifications and tests developed with the help of UniTesK technology in almost arbitrary way.

Along with implementation of the specification interface, mediator is responsible for synchronization of model state defined in specifications with the state of the implementation, which can have entirely different form. This is important to support *open state testing,* which assumes that on each step (between two calls of target methods) we can determine exactly what the current state of the components implementation is.

Fig. 3 demonstrates the complete set of the main components of UniTesK test suite architecture. Test sequence iterator component details are not presented, look at Figure 2 for them. Auxiliary components of test suite responsible for tracing and run-time test support are not showed too.

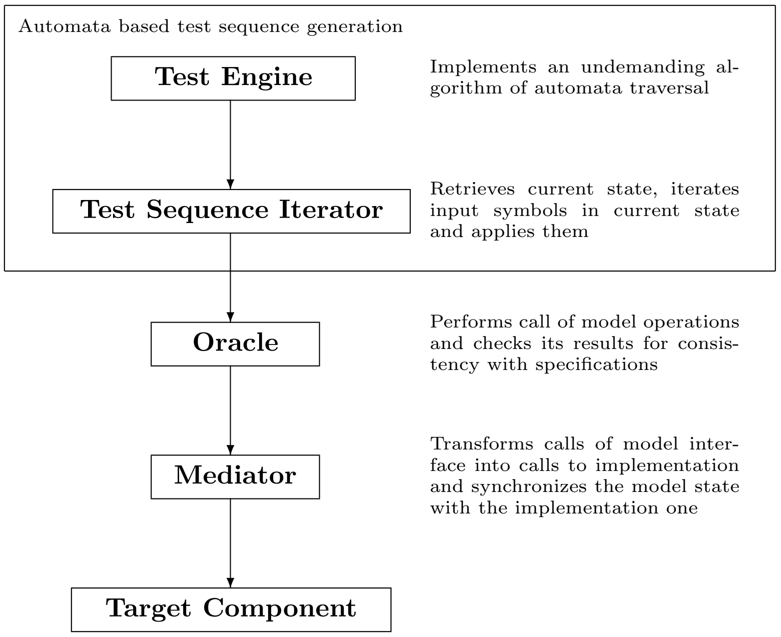
Automata based test sequence generation
Test Engine
Implements an undemanding algorithm of automata traversal
Test Sequence Iterator
Retrieves current state, iterates input symbols in current state and applies them
Oracle
Performs call of model operations and checks its results for consistency with specifications
Mediator
Transforms calls of model interface into calls to implementation and synchronizes the model state with the implementation one
Target Component

Fig. 3. Complete architecture of UniTesK test suite

Mediator components are usually written by hand. This fact increases the total size of manual work, but in return we have the possibility to develop really reusable specifications and tests, sometimes comprising the complete and ready-to-use suite for regression testing.

4 Comparison with Existing Approaches

The need of well-scalable systematic approach to industrial test development is generally recognized. The discussion of JUnit [11] testing framework in Java world shows that once more. It also demonstrates that the mere set of base classes and interfaces of the test suite components and a reasonable guide on creating tests on the base of this framework can significantly improve unit testing procedures commonly used in the industry.

UniTesK approach makes several more steps. Like JUnit, it gives test developers a flexible architecture of test suite. But in addition it determines a minimum data set required to generate all of the test suite components automatically. After that it proposes a simple representation of these data and gives tools for its automatic transformation into a ready-to-use test suite.

UniTesK test suite architecture also have many similarities with other model based testing approaches emerged during last decade, like the one used in CADP [12–14] tool or the one proposed in AGEDIS [15] project and based on GOTCHA-TCBeans [16, 17] tool. These approaches also propose a flexible architecture of test suite supporting reuse of behavior models. Both of them use FSM based mechanism of test generation. GOTCHA tool, similar to UniTesK approach, supports coverage driven test generation. AGEDIS test architecture also proposes the components similar to UniTesK mediators.

The main differences between UniTesK and CADP/GOTCHA approaches are the following.

First, these approaches use some universal modeling language. CADP is on LOTOS and some specific notation for FSM description, and GOTCHA proposes specific GOTCHA Definition Language for description of automata models along with test cases and coverage tasks. UniTesK supposes to use uniformly defined extensions of well-known programming languages, like Java, C/C++, and so on. The uniformity of these extensions is guaranteed by the coincidence of extension constructs sets. We should add special constructs to express pre- and postconditions of operations, data type invariants, and several constructs making possible implicit FSM description (specifying only the state data structure and possible input symbols for each state) in the form of test scenarios.

Second, both CADP and AGEDIS methods suppose that the single automata model expresses both the constraints on the target system behavior and is used for test case generation. This is not convenient in general case, so we often need to use different models of the target system depending on the level of test coverage we want to achieve.

UniTesK proposes strict distinction between the model of behavior used to check its correctness and the testing model used for test generation. We need only one behavior model, which is represented in our case in the form of pre- and postconditions of operations and data type invariants. One can also notice that UniTesK behavior model can be implicit, while CADP or GOTCHA always requires an executable one.

A testing model encoded in test sequence iterator component is always FSM. We can use different testing models depending on the goals of testing. But in UniTesK approach it is much more convenient, because a testing model should not include information on correct or incorrect behavior, like in other FSM based testing techniques. So, the description of such a model is more compact and easier to use and maintain.

The only approach we know that uses two different models for behavior correctness checking and for test design is used in Rational Test RealTime [18] tool for functional testing. But this tool lacks any support for automated test generation based on FSM models.

Although most elements of UniTesK test suite architecture can be found in other approaches, none of them combines all the UniTesK features and provides the similar level of flexibility and automation support.

5 Conclusion

As we see from the previous sections the test suite architecture proposed possesses such features as full support for conformance testing, test coverage monitoring and dynamic test optimization according to the target coverage criterion, support for user-developed test scenarios, flexibility, reusability of the most part of the components, high level of automation combined with the possibility to tune up the test suite in many ways, and, last but not least, rigorous theoretical base.

This set of features along with our experience in industrial testing makes sound our hope that UniTesK test development technology based on the architecture presented can bring many benefits in industrial software testing.

We can also notice that this architecture is developed in conjunction with tools for an automatic generation of all the components that are marked above as able to be generated. So, from its origin, the architecture is intended for industrial use and has been already demonstrated its capabilities in several industrial testing projects.

Just now we have tools supporting UniTesK test development for Java and C (lite version); we also developed the similar tool for testing models developed in VDM++ [19]. This tool is implemented as an add-in for VDM Toolbox. The list of research and industrial project performed with the help of these tools and more information on UniTesK technology can be found on [1].

References

1. http://www.ispras.ru/~RedVerst/
2. http://www.fmeurope.org/databases/fmadb088.html
3. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. *FM'99: Formal Methods. LNCS,* volume 1708, Springer-Verlag, 1999, pp. 608–621.
4. D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering,* 24(3):161–173, 1998.
5. M. Obayashi, H. Kubota, S. P. McCarron, L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.xopen.org/exgr/icse/icse98.htm
6. I. B. Burdonov, A. S. Kossatchev, and V. V. Kulyamin. Application of finite automatons for program testing. *Programming and Computer Software,* 26(2):61–73, 2000.
7. A. Petrenko, I. Bourdonov, A. Kossatchev, and V. Kuliamin. Experiences in using testing tools and technology in real-life applications. *Proceedings of SETT'01,* India, Pune, 2001.
8. A. K. Petrenko. Specification Based Testing: Towards Practice. *Proceedings of PSI'01. LNCS,* Springer-Verlag. To be printed.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.
10. 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,* Springer-Verlag. To be printed.

11. http://junit.sourceforge.net/
12. http://www.inrialpes.fr/vasy/cadp/
13. J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. Proceedings of the 8-th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA), 1996, pp. 437–440.
14. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. INRIA Technical Report TR-254, December 2001.
15. http://www.agedis.de/
16. http://www.haifa.il.ibm.com/projects/verification/gtcb/documentation.html
17. 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.
18. http://www.rational.com/products/testrt/index.jsp
19. http://www.ifad.dk/