UniTesK: Model Based Testing in Industrial Practice

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

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, 109004, Moscow, Russia
Tel. +7 095 9125217
Fax. +7 095 9121524
{kuliamin,petrenko,kos,igor}@ispras.ru
http://www.ispras.ru/groups/rv/rv.html

Abstract. The article presents UniTesK technology, an automated model based test construction method for use in industrial testing of general-purpose software. The approach presented includes automatic generation of test oracles from software contracts, coverage driven test sequence generation, test artifacts reuse. This work stems from the ISP RAS results of academic research and 10-years experience in industrial application of formal testing techniques [1].
Keywords: model based testing, formal specifications, finite state machine, FSM based testing.

1 Introduction

Since the very origin of software engineering developers dream of an automated technology of test development, which would be applicable for real-life software. During past decades research community proposed many approaches to its realization. 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.

Test development and testing usually requires a lot of manual work. A lot of naive attempts to automate test production do not give valuable results because they leave out of scope functional requirements to the software under test. To involve requirements in automated test construction we have to use formal representation or *model* of them. So, one comes to an idea of *model based* test development and testing. The model used should be much simpler than the system itself, lest it would require too much effort to construct. And we need to solve the contradiction between model simplicity and the complexity of tests necessary to test complex target system.

There exist a lot of model based test construction methods and tools developed in research community and each year new ones appear. But the industry has its own view on the suitability of such methods. Only a few of academic methods and tools has real-life applications. This article presents UniTesK – a model based test construction method developed with intention to bridge the gap between advanced test construction methods and techniques and tools suitable for the industry. It was designed in ISP RAS on the base of the experience obtained during several industrial software verification projects. UniTesK deals mostly with models of functionality of target systems, so the issues of non-functional model based testing are out of the scope of this article.

The next section provides description of UniTesK method starting from the general overview of the approach and proceeding to more detailed explanation of its main ideas. The third section compares UniTesK with other academic and industrial test construction techniques. The last section briefly describes UniTesK applications and possible future evolution.

2 Overview of UniTesK Technology

2.1 General Approach

We believe that only a combination of several factors can make test development method appropriate for industrial use. First, it should be supported by tools. Ideally, all the steps and activities able to be automated should be automated. Second, the method should be based on widely used and clear concepts and simple notation, which do not require education and skills more than an average software engineer has. Third, it should provide a large set of features that can make it flexible enough to be useful in various industrial contexts.

To achieve these goals UniTesK technology provides the following solutions approved by a long use in real-life software testing projects in various domains.

- The technology is based on the uniform test architecture. The architecture is developed on the idea to evolve the minimum information that should be supplied by human in general case and put it in a number of components with well-defined responsibilities. All the rest parts of test program are generated automatically. Moreover, in many situations we may provide interactive semi-automatic generation of all test components, except for *specifications* stating the criteria of software correctness.
- The well-known Design by Contract [2] approach is used as a specification technique. Software contracts can be automatically transformed into *test oracles,* components checking software correctness during testing.
- Specifications are represented in extensions of widely used programming languages. This feature makes the technology more clear for an average software developer. Training requires short time and useful results can be obtained much earlier. ISP RAS has already developed tools supporting UniTesK technology and based on the extensions of Java, C, and C#.
- Finite state machine (FSM) models are used for automatic test generation. To make the test designer work simpler, auxiliary tools provide automatic construction of FSM model required on the base of specifications and user-defined strategy for testing. Thus, usually user may not worry about the mathematical models used for test construction.
- Possible test generation strategies include coverage-driven ones, targeted to achieve a full coverage according to several available *functional coverage criteria.* Such criteria can be automatically extracted from the contracts or user-defined.
- Contract specifications can be used not only as assertions, which are placed in the code of target software. One can separate them from the code and use as a full-fledged formal representation of functional requirements. Special *mediators* are used to bind such specifications with some implementation to test their conformance. This feature opens the door for many others, for example, the following ones.
  - Specifications can be made more abstract and more close to the natural representation of requirements.
  - Correspondence between requirements (represented as specifications) and tests can be supported almost automatically.
  - Specifications remain actual for different versions of target software. Only mediators should be replaced if the change between versions does not touch the functionality, and this replacement can be supported by special tools.
  - Extensive reuse of specifications and tests becomes possible. Thus, effort once invested in the development of specifications and tests can be repaid several times.

- Co-verification software development based on concurrent development of the target system and tests for it becomes possible.
- Specifications of off-the-shelf software components functionality can be provided separately from their implementation, thus making possible functional testing adjusted for user needs and independent confirmation of component conformance.

2.2 Uniform Test Architecture

Versatility of a tool or technology, possibility to apply it in a wide variety of situations is determined mostly by the underlying architecture. The test architecture used in UniTesK was designed on the base on ISP RAS experience in testing industrial software and is intended to solve two main problems.

- How to compromise automation of test development with need for information that can be provided only by human, like software correctness criteria or testing strategy.
- How to provide a uniform solution suitable for testing in different domains and capable to embrace changes coming from several sources.

The main approach used is to define a set of components capable to perform testing for different software kinds and using different testing strategies. Then, we try to encapsulate information that can be provided only by human in a few components with clearly defined responsibilities. For each logical part of this information we try to provide suitable and simple notation.

UniTesK test architecture [3] is based on the following separation of concerns. The problem of checking software behaviour correctness after some single external action is separated from the problem of binding abstract model and specific implementation and from the problem of test sequence construction to provide required testing quality. To handle each of these problems some support is provided.

To check the correctness of target software behaviour *a test oracles* are used. We use general oracles capable to assess behaviour of the target system in response on an arbitrary single action. Such an oracle can be easily generated from the software contract in the form of pre- and postconditions of interface operations and invariants of interface types. Each possible external action is modelled as a call of some operation with some arguments. See the next subsection for details of specification technique used.

To construct a test sequence we use a finite state machine (FSM) model of the target system. FSMs are familiar for software developers and capable to model the behaviour of almost any software. To be able to test concurrency and distributed systems we use a kind of *input-output finite state machines* (IOFSMs) [4]. FSM model is specified with the help of *test action iterator* component, and test sequence is generated on-the-fly, during the testing, by a kind of graph traversal algorithm encapsulated in *test engine.* We use term *test scenario* for human-readable FSM model description, which can be developed manually or interactively generated from specification and test strategy chosen. Test action iterator is automatically generated from test scenario. Test engines are provided as library components, which can be chosen for use depending on the developer goals. More details of FSM model construction are provided in the subsection *Test Construction Methods* below.

To make use of specifications written on more abstract level then the target software is we provide *mediators.* Each mediator component binds some specification with implementation of the corresponding functionality in the target system. It provides transformation

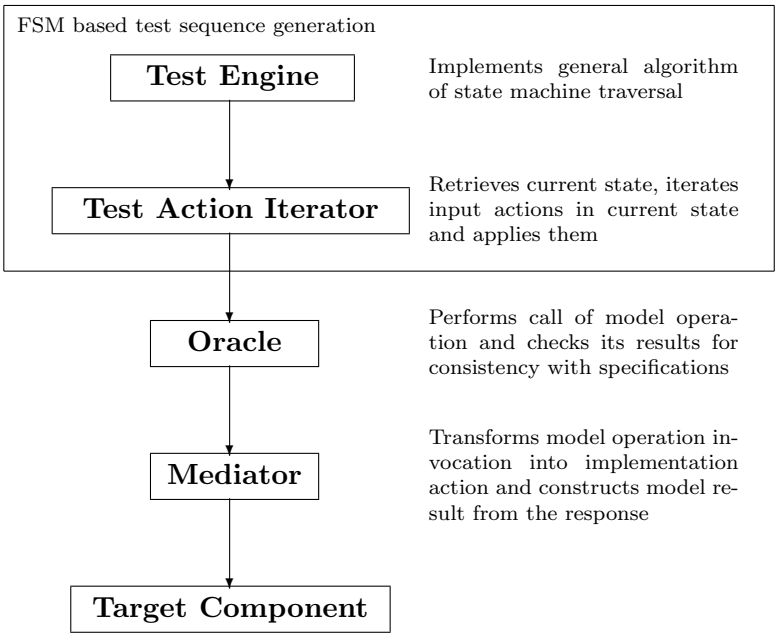

Fig. 1. UniTesK test architecture.

of model operation call into corresponding external action on the target system and then transform the implementation response into the result of model operation. Due to specifics of implementation environment a lot of auxiliary work is required to be performed by mediator - exception handling, tracing, etc. Code to implement it can be inserted automatically into the description of transformation procedures provided by user or constructed on the base of correspondence set between model operations and implementation actions.

Fig. 1 summarize the information given above and presents the main components of UniTesK test architecture. In addition to them test system contains some auxiliary components responsible for tracing, synchronization of model and implementation states, test execution support, etc. They do not depend on the software under test or testing strategy chosen.

2.3 Specification Technique and Functional Coverage Criteria

UniTesK technology provides automatic generation of test oracles from formal specifications represented as software contracts. Target system is described as a set of components, each of which has a number of interface operations that can be invoked by the environment. A component may have state that affect the externally observable behaviour of its operations and can be changed by them.

An operation is described with the help of its precondition, stating the responsibilities of the clients, and its postcondition, defining the properties of the operation result and resulting state, which should hold after operation execution if the precondition holds before it and the system behaviour is correct. Operation may have parameters of some types. These types and types of the abstract state data are called *interface types* and are described in terms of their data structure and data integrity constraints called *invariants.*

Software contracts are chosen as a main specification technique because they are rather simple, applicable for software description in many domains, can be made either abstract

or detailed depending on the developer goals, and have clear structure close to the structure of requirements. An ordinary software engineer can be trained to comprehend and use software contracts with little effort.

UniTesK uses the structure of contracts to define *functional coverage criteria,* which are necessary to assess the quality of testing in terms of requirements. To facilitate automatic extraction of such criteria postconditions are to have the following form. The control flow graph of a postcondition after contraction of cycles and exception handling blocks is acyclic. Specification developer should define a cut set of this graph with the help of `branch` operators, i.e., put these operators in the postcondition code so, that each path from the entry point to an exit of the postcondition contains exactly one such edge. Each `branch` operator define a subdomain of the operation specified, called *functionality branch.*

Predicates corresponding to functionality branches can be automatically calculated and used to measure the coverage obtained during the testing. These predicates are also translated into conjunctions of branching conditions of the postcondition code and into disjunctive normal form (DNF) to provide two more detailed coverage criteria. The first one corresponds to different paths from the entry point of the postcondition to some `branch` operator, and the second corresponds to disjuncts in the DNF, similar to well known MC/DC source code coverage criterion [5]. User may also define additional coverage goals.

2.4 Specification Notation

UniTesK tools support specifications in extensions of widely used programming languages. This makes the technology simpler to use for developers familiar with such a programming language.

Formal specification languages are usually more abstract and expressive than usual programming languages, but most modern programming object-oriented languages can be easily augmented with libraries representing useful abstract concepts.

Formal specification languages have formal semantics, so specifications are unambiguous. The choice of programming language as specification notation seems to dismiss this benefit. But instead we obtain two serious advantages. The first one is that we have no need to bind specification language with implementation one, which is very hard and often requires being an expert in both. The second one is that we can construct a model of the target system more easily, because there is no paradigm difference between implementation and model languages.

2.5 Test Construction Methods

UniTesK uses FSM models represented as *test scenarios* to generate sequence of test actions on-the-fly. Test scenario defines what is considered as a state of an FSM and what model operations with what arguments should be called in an arbitrary state. During testing test engine performs a transition tour or some other "exhaustive" path on the state machine defined by scenario and so generates the test sequence.

This method guarantees that the state of target system is changed only by operation invocation and only reachable states are considered. Only arguments of model operations should be iterated in so far that the test can obtain the coverage needed. If some coverage criterion is set as a goal, the arguments that add nothing to the coverage value achieved are filtered out. Thus, test generation is coverage driven.

Test scenario represents state machine in *an implicit form,* i.e., states and transitions are not explicitly enumerated and transitions ends are not specified. Instead, scenario provides state identification procedure, input symbol iteration procedure depending on state, and the way how to execute an input symbol. Although such representation is not usual, it provides short and simple descriptions for rather complex models.

Test scenario may define state structure different from the state of behaviour model described in specifications. Thus, when we need to take into account some implementation specifics, we can provide the test generation mechanism with the related data without changes in specifications. From the other hand, we may abstract of some details of model state, so decreasing dramatically the state space without loss of details in the constraints to be checked. The corresponding technique called *factorization* is formally described in [6].

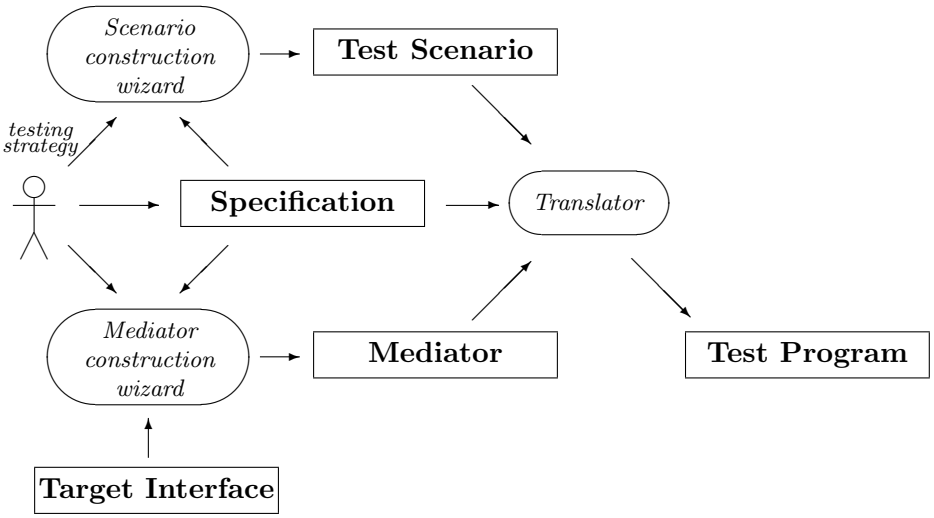

Fig. 2. UniTesK test development process.

Test scenarios may be developed manually, but in most cases they can be generated automatically with the help of auxiliary interactive tool, *scenario construction wizard,* which asks what specification to use, what implementation to test, which coverage criterion to set as a goal, and how to iterate arguments of operations.

Test scenarios are usually written in specification terms and so are be highly reusable. They also can be reused with the help of inheritance.

To test concurrent and distributed software UniTesK uses special test engine generating pairs, triples, or n-tuples of parallel input actions and slightly modified form of specifications. In addition to specifications of interface operations they may contain *asynchronous response specifications.* The corresponding state machine models used for test generation are very similar to IOFSMs. Correctness checking is performed by construction of permissible sequences of actions provided and responses obtained. If such a sequence cannot be constructed, it is considered as a discrepancy between specification and implementation.

2.6 Binding Tests with Target Software

UniTesK specifications can be decoupled from implementation with the help of mediators. Mediators make possible more abstract and reusable specifications, and thus, tests.

Mediator may be developed manually in an extension of target programming language, but in most cases it can be generated with the help of auxiliary interactive tool, *mediator construction wizard,* which asks for specification and implementation components to bind and correspondence between their operations. For each operation one should also provide procedures to transform model arguments into implementation ones, and implementation responses into model result.

Fig. 2 presents the general scheme of test development used in UniTesK. When the development of specifications, test scenarios, and mediator is finished, they are translated from the extended language to the target language and give a ready for use test program as a result. Test execution produces test trace and several reports presenting coverage levels achieved according to each criteria defined, the structure of FSM used for test generation, failures detected, etc.

Test reports can help greatly in the analysis of failure causes, because they contains additional information on failure type, the precise condition with check is failed, information on the functionality branch, in which the failure occurs, some information on location of failure if it is in a component of test program, etc. They also help to evaluate the testing quality in terms of requirements checked.

3 Comparison with Existing Approaches

It seems that for each of UniTesK features one can find a test construction approach developed in the last ten years and having the same feature, often in more elaborated form. Nevertheless, none of the existing approaches has the same set of features.

In this short review we pay most attention to the test construction methods supported by tools and oriented to the use in the industry. A lot of interesting academic methods without sufficient automation features are left outside of its scope.

Flexible, well-scalable, and systematic approach to industrial test development has to be based on a well-defined architecture of tests. However, most part of existing techniques do not pay much attention to the test architecture and produce tests in the form developed during the long period of manual test development – a set of test cases. The success of JUnit [7] testing framework in Java world demonstrates that reasonable test architecture can improve test development even with minimum automation.

UniTesK approach goes further. It gives test developers flexible test architecture and in addition it determines a minimum data set required to generate all the test components automatically, suggests a simple representation of these data, and gives tools for its automatic transformation into a ready-to-use test suite. The similar facilities are incorporated in commercial test development tools such as Rational Test RealTime [8] and Parasoft JTest/JContract [9].

In contrast with tools using specification extension of programming language to represent software contracts, such as JTest or various tools based on Java extensions (see reviews of them and references in [10, 11]), UniTesK provides essential support for coverage driven test development and allows specifications to be more abstract than the code under test and decoupled from it.

The large group of academic test construction tools and techniques are based on various kinds of transition systems (EFSMs, IOFSMs, Communicating EFSMs, Statecharts) as models of software under test. They are mostly appropriate for telecommunication software development where requirements are usually represented in such a form or in related specification languages like SDL, Estelle, or LOTOS. These tools use *test purposes*

describing the part of behaviour to be tested in a test case (see [12–14]) or, similar to UniTesK, may provide test generation intended to obtain some coverage (see [15, 16]).

GOTCHA [17] and AsmL Test Tool [18] are the only two known to authors, which can use factorization of state spaces [19] similar with UniTesK tools. The most part of other transition-system-based tools are prone to space explosion problems or try to reduce the number of states with the help of some specific methods. Both GOTCHA and AsmL Test Tool similar with UniTesK utilize the idea of using different models for description of software functionality and for test sequence construction. However, the only kind of factorization available for GOTCHA is projection (leaving some state fields outside of consideration). Both that tools differ from UniTesK tools by using specific formal languages for model representation.

4 Conclusion

A test development technology should posses some 'critical mass' of features so that the industry can actually benefit from it. RedVerst [1] group of ISP RAS proposes UniTesK test development technology as a candidate. UniTesK is a successor of KVEST [20, 21] test development technology developed ISP RAS for Nortel Networks. It is based on a ground experience in specification based testing obtained from 10-years work in several software verification projects. The total size of software tested in these projects is over half a million lines of code. UniTesK tries to keep all the positive experience of KVEST usage, but introduces some improvements in flexibility of technology, heightens the reusability of various artifacts, and lessens the skills required to start using the technology in an effective way.

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 in UniTesK is usually much more simple than the one required to develop the similar test suite in traditional approaches. The most tedious and mechanical parts of work are performed automatically.

Just now we have tools supporting UniTesK test development for Java and C; we also have developed the tool for testing .Net components based on C# extension [22, 23]. The existing tools were successfully used in testing several IPv6 implementations, to model distributed debugger for mpC language, and to test themselves. The complete list of research and industrial project performed with the help of UniTesK technology can be found on [1].

References

1. http://www.ispras.ru/~RedVerst/
2. Bertrand Meyer. Applying 'Design by Contract'. IEEE Computer, vol. 25, No. 10, October 1992, pp. 40–51.
3. 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.
4. 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.
5. J. J. Chilenski, S. P. Miller. Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, pp. 193–200, September 1994.
6. I. Burdonov, A. Kossatchev, and V. Kulyamin. Application of finite automatons for program testing. Programming and Computer Software, 26(2):61–73, 2000.
7. http://www.junit.org/index.htm

8. http://www.rational.com
9. http://www.parasoft.com
10. M. Barnett, W. Schulte. Contracts, Components, and their Runtime Verification on the .NET Platform. Technical Report TR-2001-56, Microsotf Research.
11. L. Baresi, M. Young. Test Oracles. Tech. Report CIS-TR-01-02. Available at http://www.cs.uoregon.edu/~michal/pubs/oracles.html
12. 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.
13. 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.
14. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. INRIA Technical Report TR-254, December 2001.
15. 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). with verification techniques. International Workshop on Protocol Test Systems (IWPTS), Evry, 4-6, September, 1995.
16. 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.
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. 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.
19. G. Friedman, A. Hartman, K. Nagin, T. Shiran. Projected state machine coverage for software testing. Proc. of ISSTA 2002, Rome, Italy. Jul 2002.
20. 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.
21. http://www.fmeurope.org/databases/fmadb088.html
22. http://www.atssoft.com
23. http://unitesk.ispras.ru