KVEST: Automated Generation of Test Suites from Formal Specifications

Igor Burdonov[1], Alexander Kossatchev[1], Alexander Petrenko[1], and Dmitri Galter[2]

[1]Institute for System Programming of Russian Academy of Science (ISP RAS), Moscow, Russia
{Igor, Kos, Petrenko}@ispras.ru
[2]Nortel Networks, Ottawa, Canada
Galter@nortelnetworks.com

Abstract. KVEST - Kernel VErification and Specification Technology - is based on automated test generation from formal specifications in the RAISE specification language. The technology was developed under contract with Nortel Networks. As of 1999, the methodology and toolset have been applied in three industrial project dealing with verification of large-scale telecommunication software. The first project, the Kernel Verification project, gives its name to the methodology and the toolset as a whole. Results of this project are available from the Formal Methods Europe Application database [13]. It is one of the biggest formal method application presented in the database. This paper provides a brief description of the approach, comparison to related works, and statistics on completed projects.

1 Introduction. History of KVEST Development and Use

In 1994, Nortel Networks (Bell-Northern Research, Northern Telecom and Nortel are the former names of Nortel Networks) proposed Institute for System Programming of Russian Academy of Science (ISP RAS) to develop a methodology and supporting toolset for automation of conformance testing of Application Program Interfaces (API). The real-time operating system (OS) kernel was selected as a first practical target for the methodology. ISP was to rigorously describe software contract definition of the kernel and produce test suites for the kernel conformance testing.

The success of this work would allow Nortel to automate conformance testing for the next OS kernel porting and for the new release of the OS. In addition, Nortel would be able to improve its product software structure as a whole, since during software contract definition, ISP promised to establish minimal and orthogonal set of interfaces for the OS kernel.

ISP organized a joint group of researchers and developers. The team members had rich experience in operating system design, real-time systems, compiler development, and formal specification use in a systematic approach for design and testing [2, 9,10].

During the first half year, ISP developed the first version of the Kernel Interface Layer (KIL) contents and conducted a comparison analysis of the available specification methodologies. The KIL contents were approved with slight modifications. RSL (RAISE Specification Language) [11,12] was selected as the most suitable specification language.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 608-621, 1999.

During next half year, ISP developed a first draft of the specification and test generation methodology and developed a prototype version of the specifications and test oracle generator. The prototype demonstrated the possibility to use formal specifications in industrial software development. Implicit specification was selected as the main kind of specification. The base principles of test coverage analysis were established. KVEST methodology uses a modification of FDNF (Full Disjunctive Normal Form) criteria for the partition of input space, test coverage and test strategy.

During the second year, product version of the specifications and tools for test generation and execution were completed. From mid-1996 to the beginning of 1997, the majority of the test suites were produced and the OS kernel was successfully tested. The number of detected latent errors exceeded expectations.

2 Terms and Basic Notions

Let there be some software system containing the functionally closed set of procedures. We need to determine the elements and functional specifications of its external interfaces, constituting the software contract, and to develop a set of test suites suitable for conformance testing of the software contract implementation. Since the elements of the software contract are procedures, we can say that this is in fact API (Application Programming Interface) testing. Here and below we will say that the API consists of a set of procedures. There are other kinds of API entities like operations, functions, methods (in C++), subroutines in Fortran, etc. We consider all the terms synonyms and will use the term *procedure* in this paper.

Let us note that we are not talking about testing some specific implementation of the software contract. It was important to build a methodology allowing to verify correctness of the software behaviour without introducing any extra limitations on the internal structure of the implementation. To stress this very important requirement, we call our specifications implementation independent.

Let us introduce some terminology. The whole software system used in the process of test execution and communicating with the system under test (SUT), is called the *test harness*. Sometimes, instead of SUT, we will use the term *target system*. The main part of the test harness is the *test drivers*. To enable the functionality of the test drivers, we need a run-time support system, which is often called a *test bed*. The test drivers are usually developed with the SUT specifics in mind while the test bed is independent of the SUT functionality.

We consider two levels of test drivers. A *basic driver* is a test driver for some target procedure that performs the following actions:

- checks that pre-conditions for the target procedure hold for a given tuple of input parameters;
- calls the target procedure with a given tuple of input parameter and records corresponding output parameters;
- assigns a verdict on the correctness of the target procedure execution results;
- collects information necessary to estimate the test coverage or investigate reasons for a fault.

A *script driver* is a test driver for some target procedure, or a set of target procedures, that performs the following actions:

- reads test options;
- generates sets of input parameters based on test options;
- calls a basic driver with some set of input parameters;
- does extra checking, if necessary, of the correctness of the target procedure execution results and assigns a verdict;
- checks if desired test coverage is complete and if not, continues to generate sets of input parameters and call the basic driver with this tuple.

A *test plan* is a program that defines the order of script driver calls with the given test options and checks the script driver call conditions and termination correctness.

Besides the set of basic and script drivers and test plan interpreter, the test harness also contains a repository and tools for test plan execution and querying the results kept in repository. The repository contains information about all test executions, code coverage metrics for different procedures with different test coverage criteria, and all situations when test drivers assigned a negative verdict.

3 KVEST Overview

KVEST methodology consists of the following steps:

- Software contract content definition;
- Specification development;
- Test suite production;
- Test execution and test result analysis.

3.1 Software Contract Content Definition

Goals:

- to provide the minimal and orthogonal interface;
- to hide internal data structures and implementation details.

Following these goals we can minimize the restrictions on the possible implementation solutions and the knowledge needed to use the software, and make it possible to develop long-term living test suites for conformance testing.

3.2 Specification Development

Goals:

- to rigorously describe functionality;
- to provide an input for test generation.

Based on the specification, KVEST can fully automatically generate the basic drivers.

3.3 Test Suite Production

Goals:

- to develop the so-called “manually developed” components (MDC) of test suites;
- to generate test suites.

Most of the MDCs are convertors between model and implementation data representation, test data structure initiators, and iterators. Based on the MDCs and the specifications, KVEST generates test suites as a whole. No customization is required once the generation is complete.

3.4 Test Execution and Test Result Analysis

Requirements of the tool for test execution and analysis are as follows:

- to automate test execution;
- to collect trace data and to calculate obtained test coverage;
- to provide browsing and navigation facilities;
- to recover the “incremental testing” target system after a fault or a crash, and to continue test execution from the point of interruption.

3.5 Specification Approach

Let us consider the specification approach in more detail. KVEST uses model-oriented specification in implicit (pre- and post- condition) form as the main form of specification. More exotic RAISE features like axiom, algebraic specifications and channels are not used in the formal specification but are used in semi-formal considerations and explanations.

Concrete examples of implicit specifications are shown below.

3.6 Testing Approach

To develop a test driver we have to solve three problems:

- how to generate an oracle, i.e. a program that assigns a verdict on the correctness of outcome for the target procedure;
- how to estimate completeness of the test coverage;
- how to enumerate combinations of the test input data.

Test oracles are very similar to post-conditions. Both functions are Boolean. They have the same parameters, and return True if the target procedure produces correct

result and False otherwise. So, the generation of test oracles is quite feasible once we have the post-conditions.

Test coverage criterion is a metric defined in terms of implementation or specification. The most well known test coverage criteria in terms of implementation are:

- C1 - all statements are passed;
- C2 - all branches are passed.

In case of specification use for test coverage criteria definition, the so called domain testing approach is used. The whole input space is partitioned into areas. Each area corresponds to a class of equivalence. The partition could be derived from the specification that describes requirements on input and properties of outcome for target procedures. Both the requirements and the properties are clearly represented in pre- and post-conditions of formal specifications in implicit form. So, based on the implicit specification, we can successfully solve the problem of test coverage estimation.

There are sceptics who think that full coverage of domains, even including interesting points like a boundary layer, does not guarantee good coverage of the implementation code. Our experience shows that the average percentage of KVEST test coverage is 70% to 100% of statements in the implementation.

We distinguish two levels of the test coverage criteria. The first one is the coverage of all branches in post-conditions. The second one is the coverage of all disjuncts (elementary conjunctions) in the Full Disjunctive Normal Form (FDNF) representation of the post-condition while taking into account the pre-condition terms. KVEST allows partitioning in terms of specification branches and FDNF to be made fully automatically. One of the most difficult problems is the calculation of accessible FDNF disjuncts and removing the inaccessible FDNF disjuncts. This problem is solved in KVEST by a special technique in pre-condition design.

Monitoring of obtained test coverage is conducted on the fly by script drivers. Based on this data, the script driver may tune testing parameters and/or testing duration.

4 Test Generation Techniques

4.1 API Classification

First we should consider a classification of API. The classification determines the choice of one of the test generation techniques applicable to a procedure interface or interface of a procedure group.

We consider five main classes of API and some extensions of classes including interfaces tested in parallel and expected exceptions.

The classes are organized hierarchically. The first class establishes the strongest requirements. Each following class weakens the requirements. The requirements for the five classes are as follows:

Kind 1. The input is data that could be represented in literal (textual) form and can be produced without accounting for any interdependencies between the values of diffe-

rent parameters. Such procedures can be tested separately because no other target procedure is needed to generate input parameters and analyze the outcome.

Examples of interdependencies will be shown below.

Kind 2. No interdependencies exist between the input items (values of input parameters). Input does not have to be in literal form. Such procedures can be tested separately.

Example: Procedures with the pointer type input parameters.

Kind 3. Some interdependencies exist, however separate testing is possible.

Example: A procedure with two parameters, the first one is array, the second one is a value in the array.

Kind 4. The procedures cannot be tested separately, because some input can be produced only by calling another procedure from the group and/or some outcome can be analyzed only by calling other procedures.

Example: A procedure that provide stack operations and that receives the stack as a parameter.

Kind 5. The procedures cannot be tested separately. Part of the input and output data is hidden and user does not have direct access to data.

Example: Instances of OO classes with internal states; a group of procedures that share a variable not visible to the procedure user.

Exception raising extension of API classes. The specific kind of procedures that raise exceptions as a correct reaction to certain inputs.

Example: A procedure that is supposed to raise an exception after dividing by zero. If zero received as an input parameter, then this procedure must not return any return code.

4.2 Script Driver Scheme. Example of Kind 5

The above taxonomy is a good basis for the classification of test generation techniques. Kind 1 allows full automation of test generation. All other kinds need some additional effort for MDC writing. The effort gradually grows from kind 2 to kind 5. The extensions require more effort than the corresponding kinds themselves.

Complexity and effort of the MDC development is caused by the complexity of script driver writing and debugging. Below we consider only one scheme of script drivers used for kind 5 API testing. All script drivers have similar structure. The main distinction is the distribution between automatically generated components and MDCs. The kind 1 script driver is generated fully automatically, kind 2 script driver - almost automatically and so on.

The script driver is a program that is composed and compiled by KVEST. The general scheme of the script driver is defined by a formal description called a *skeleton*. The skeletons are specific to each kind of API. Each script driver consists of declarations and a body. The declarations are generated automatically based on the list of proce-

dures under test and their specifications. The declarations include import of the procedure under test and its data structure definitions and/or import of all data and types used in the specifications.

The body of a script driver begins with the script driver option parsing. The options - parameters of the script driver as a whole - determine the depth of testing (i.e. the level of test coverage criteria, and some specific data like interval of values, duration of testing, etc.).

Before testing starts some initialization is required. For example, before testing write/read procedures we have to open a file. Such initializations are written manually. After initialization is finished, the main part of the script driver begins.

The kind 5 script driver realizes a general algorithm for traversing an abstract Finite State Machine (FSM). The goal of the algorithm is to pass all states and all possible transitions between the states. Each transition corresponds to an execution of a procedure under test.

The aforementioned data are the data used in the formal specification, the so-called *model data*. The algorithm of a script driver is related to the specification and does not depend on the implementation details outside the specification.

The most interesting aspect of the script driver algorithm is the absence of direct descriptions of the abstract FSM. Direct specification of the FSM requires extra effort, so KVEST avoids this. There are some attempts to extract FSM specification from the implicit specification [1,7]. However, no one yet can provide a fully automated way for such extraction.

Instead of a direct specification of FSM, KVEST uses its indirect, virtual representation. Such representation consists of a function-observer and a function-iterator. The function-observer calculates on the fly the current state in the abstract FSM. The function-iterator selects the next procedure from the group and generates a tuple of the input parameter values for this procedure.

Let us consider the kind 5 script driver algorithm in more detail. For example, suppose we are testing a procedure group. Say, we have passed several FSM states, which means we have called some target procedures. Now we are going to make the next transition. This elementary cycle of testing consists of the following steps:

- Calling a function-iterator that selects the next procedure from the group and prepares a tuple of input parameter values for this target procedure.
- If the iterators have managed to generate a new and correct tuple without violation of pre-conditions, then the script driver calls a corresponding basic driver with the tuple as actual parameters.
- When the basic driver returns, the control script driver checks the verdict assigned by the basic driver.
- If the verdict is False (an error has been detected), the script driver produces corresponding trace data and finishes.
- If the verdict is True (the elementary test case passed), the script driver calls the function-observer that calculates a current state, logs the state and transition and continues to traverse FSM.

Thus, we obtain all possible states and test the procedures with all needed sets of input parameters. FSM is used here as a guideline to pass through all states the needed number of times.

4.3 Test Suite Composition

Let us come back to the issue of the composition of MDCs and automatically generated components. Script drivers are composed following the requirements of the corresponding skeletons. Overall, we need five skeletons for serial testing of API kinds 1 through 5; one skeleton for parallel testing, and five skeletons for exception raising testing. Based on a corresponding skeleton and the list of target procedures and specifications, KVEST generates the script driver template. For kind 1, this template is a ready-to-use program. For the other kinds, the template includes several nests with default initiators and iterators. If a test designer does not need to add or improve anything in the nests, the template can be compiled and executed. This situation is typical for a kind 2 API. For other kinds, the test designer usually has to add some specific initiators and iterators. In any case, he or she should define FSM state observer for the script drivers of kinds 4 and 5.

The basic drivers invoked by the script drivers are generated fully automatically. The only MDCs called from basic drivers are data converters. As mentioned above, the converters transform the model data representation into the implementation representation and vice versa. A model representation is distinguished from the implementation one by the level of abstraction. For example, models may use “infinite” representation of integers, maps, relations, and other data structures suitable for specification. Sometimes model representation is very similar to the implementation one. In this case, such transformation is done by standard translation algorithm of the specification language into the implementation language.

KVEST uses implementation language independent test generators. All generators use only RSL texts as input and output. The only component written in the implementation language is the data converters. This component is out of the scope of the test generators. So, the test generation process first produces a full test suite in RSL, and then the RSL source is compiled into the implementation language. To port a KVEST test suite from one implementation language platform to another, a user should rewrite the data converters and provide RSL to the implementation language compiler, as well as a run-time support system with test bed functions.

5 Project Result Discussion and Conclusions

5.1 KVEST Applications and Statistics

As of early 1999, three projects have been completed using the KVEST technology. The first two were pursuing only code verification [3], while the third one included the re-design of old code and development of the new implementation. In all three projects, formal specifications for software contracts and test suites for conformance testing were developed.

The research on KVEST technology was conducted starting in the middle of 1994. The first year was spent on selecting the specification language, development of specification methodology principles allowing to use it in industrial-strength projects, test methodology principles and specification based test generation. At the end of the first year of research, prototype specifications of one fifth of the real OS kernel and prototype of one of the main technology tools (generator of basic drivers with test oracles) were developed.

In September 1995, works on the production version of the toolset started. Also, development of industrial specifications and the complete automated testing system was begun.

Below we provide a brief description of the three completed projects and some statistics.

A. Kernel Verification Project (Product Phase).
Target software: OS kernel of the large-scale telecommunication system.
Time period: September 1995 - February 1997.

Goals:

- to define the orthogonal and minimal interface layer of the OS kernel;
- to conduct reverse-engineering and develop formal specifications of the interface layer in RSL;
- to develop a toolset for the automated test generation and test execution;
- to generate test suites for the conformance testing of the OS kernel.

B. Feature Processing Environment (FPE) Verification Project.
Target software: Base level of call processing system;
Time period: August 1997 - December 1997.

Goals:

- to conduct reverse-engineering and develop formal specifications;
- to generate test suites.

C. Queue Utilities Re-design and Verification Project.
Target software: Management of tree-like store for queues with different disciplines.
Time period: August 1998 - December 1998.

Goals:

- to conduct reverse-engineering and develop formal specifications;
- to re-design the legacy implementation;
- to generate test suites.

Table 1: KVEST methodology statistics

Project	Size of target system in KLOC	Specification in KLOC	Generated test suites in KLOC	Effort in man months	Personal performance in KLOC per man month
Kernel Verification	200	50	900	108	1.8[a]
FPE verification	35	15	150	16	2.2
Queue utility redesign and verification[b]	9	2	90	9	1.0

a. includes tool development
b. includes re-design and re-implementation of the software in addition to analysis and verification. This explains the additional effort.

5.2 Related Works

In this section, we will discuss the related systems that, on one hand, build the verification process on the formal specifications, and, on the other hand, propose a rather generic technological scheme. We will not discuss the systems that propose the solutions to the specific problems that have certain theoretical value but fail attempts to introduce them in the process of verification of the industrial-strength software.

ITEX - Interactive TTCN Editor and eXecutor [14].
ITEX is a test environment for communicating systems. It includes a TTCN and ASN.1 analysis and design tool, a test simulator and support for generation of complete Executable Test Suites (ETS). Here is a brief ITEX functionality summary:

- A Test Suite is made up of Test Cases in form of tables;
- ITEX provides a set of highly integrated tools for development and maintenance of Abstract Test Suites (ATS) written in TTCN;
- ITEX supports the following phases of Test Suite development: Test Case Generation, Editing, Verification, Validation and Execution.

This toolset is well integrated with SDT, an environment for design of SDL specifications. Test suites described with TTCN can be transformed to the form that allows testing both implementation in some programming language and specification in SDL.

The main shortcoming of this approach in context of our research is that it is unsuitable for API testing. TTCN does not permit declaration of pointers and other software entities that do not have textual (literal) representation.

A very serious limitation of SDL-like specifications is their explicit form. This means that it is quite easy to build models and prototypes based on them but it is very difficult to develop a system of constraints that define the union of all possible implementations. Implicit specifications overcome this problem.

ADL/ADL2 [15].
This approach is the most similar to the our group's works. From formal specifications, ADL generates test oracles and skeletons for building test drivers and documentation. A not very fundamental, but still interesting difference between ADL and KVEST is that it uses not one of the popular specification languages but extensions of C and C++ languages. In the Kernel Verification project, SPP, a similar extension of the target language, was developed during the prototyping phase. A similar kind of extension was also proposed by Liskov and Guttag[6]. There are ideas on extensions of Java and other object-oriented languages aimed at developing software in "Design-by-Contract" fashion [5]. However, despite the obvious advantages of better acceptance of such languages in the software engineering community, the concept, not to mention the common notation, is still far in the future.

The difference of the KVEST results compared to the ADL results can be explained by the difference in the range of API classes for which these methodologies can provide means for specification and automatic test generation. ADL provides adequate tools for test generation automation only for procedures whose parameters allow independent enumeration and allows testing procedures one by one. In KVEST classification, these are APIs of the first and second kind. This means that procedures with dependent parameters, procedures that require testing in a group, e.g., "open - close", or those that require testing in parallel mode, e.g., "lock - unlock", or "send - receive", are omitted. Besides, ADL does not recognize the first kind of API that permits automatic generation of the complete test suite including test case parameters and test oracles.

An interesting part of ADL is its capability to generate the natural language documentation. It is important to note that the same mechanism is used for both documentation of the target system and documentation of the test suites. It seems that the ADL authors made a conscious choice of not using any of the technologies from the Natural Language Generation field. It is easy to explain in pragmatic sense, however, it does not mean that modern natural language generation methods cannot help in the generation of the software documentation. KVEST capabilities in documentation generation are implemented in the prototype version. Still, as opposed to ADL, KVEST uses computer grammar and English dictionary for analysis and generation of natural language fragments that allows it to reduce the number of natural language errors and make the text more readable without any manual work.

ADL2 went further than KVEST in its capability of specification and testing of OO classes. This KVEST shortcoming can be explained by a corresponding RSL weakness. Extending KVEST to OO software verification is a task for 1999.

Using Test Oracles Generated from Program Documentation [8]
This work is research and as such it can not be considered a technology ready to use in industry. The main interest in this research is the analysis of difficulties and limitations that, in the authors' opinion, prevent the wide use of formal specifications during automatic testing of industrial software. D. Peters and D. Parnas and us arrived to the common understanding that those are the key problems in the task of testing automation based on formal specifications. KVEST continued research in this direction and proposed a technological scheme for partial automation of test suite development for all kinds of API.

Formal Derivation of Finite State Machines for Class Testing [7].
This work is also at the research stage. At the same time, this work is interesting in the sense that it proposes a scheme for organization of procedure group testing similar to the scheme used in KVEST. Object-Z is used as a specification language and C++ as a programming language. The task is stated as follows: to built test suites to verify conformance of the implementation to the specification using formal specifications of the methods for a class. As a test coverage criterion, the union of two criteria is used: to cover all equivalency classes that represent the areas obtained as a result of partition analysis, and then, to check results on or near the boundaries.

The authors of this work do not try to solve the problem of complete automation of test generation. Nor do they attempt to support any elements of the preparation phase with tools. Still, all the steps are described in a very systematic way and can be boiled down to various specification transformations.

Partition and boundary analysis is done manually according to the methodology proposed by the authors. In a similar way, they build the specification of oracles. Oracles, once compiled into C++, call target procedures and verify the conformance of the results to the specifications.

The most interesting part is the testing scheme which is a framework that dynamically generates test sequences of procedure calls. The framework is controlled by the FSM description which represents an abstraction of the state transition graph of the test class. The authors describe the methodology of building specifications for the classes of states and transitions between them while considering the problem of exclusion of inaccessible states.

The theoretical weakness of this approach is that it does not try to come up with a formal methodology to build transformation specifications. It is obvious that serious problems will be encountered when attempting to apply this method recommendations to the specifications of real-life complexity. In practical sense, it is clear that the process of test derivation from the specifications is mostly manual activity which limits its applicability to industrial software.

The main difference between KVEST and this work is that during testing KVEST does not need the full description of the Finite State Machine that models the states of the system under test. Instead, KVEST proposes a universal algorithm that can dynamically provide all accessible states and all possible transitions between state pairs.

5.3 Kernel Verification Prospects

The most effort-consuming work in the KVEST verification process is the reverse-engineering and specification development. This work is currently being automated. The basic idea of the approach is gradual *upwarding* of data representation in defined implementations. Upwarding is increasing of the level of abstraction [4].

During 1998, ISP conducted initial work in natural language documentation generation. The result of the experiment was a prototype demo that had been presented at the ZUM'98 conference (Berlin, September 1998).

The prototype synthesizes UNIX-like "man"-page documentation based on formal and informal components of specification. The prototype demonstrated the possibility of generating actual documentation. The consistency of the documentation is checked by means of a test execution generated from the same source, the RSL specification. This work is currently being continued to extend the variety of documentation forms and to produce more fluent natural language.

5.4 Conclusions and Future Works

KVEST experience shows that formal methods can be used in industrial software production. Level of complexity and size of KVEST applications support this thesis.

An interesting fact was derived from KVEST experience. Errors detected in KVEST projects are distributed into three categories of almost equal size. One third of errors is detected during reverse-engineering and formal specification development. Another third is detected during test execution. The analysis of the test execution results allows the detection of the typical situation that cause the wrong behaviour. Based on the analysis during repeat code inspections, we detected the last third of errors.

Note 1. Code inspection in composition with formal specification, so called "formal code inspection", can detect up to one third of the errors. Note that these errors are the most dangerous.

Note 2. Code inspection can not replace testing. Up to two thirds of errors are detected during and after testing.

Note 3. Testing is necessary to develop correct specifications. During script driver debugging, up to a half of all detected errors were the errors in the specifications. Thus, formal methods must be supported by technologies that compare formal method results with actual software by executing the target software.

The next consideration is related to the distribution of error causes. Up to one third of the errors were caused by the lack of knowledge on pre-conditions and some details of the called procedures' behavior. Because pre-conditions and variants of possible procedure behavior were not described sufficiently, users could call procedures in the wrong situations and not expect strange procedure reactions. Establishing software contracts and systematic description of the API allows the number of errors to be decreased by up to 30%.

Acknowledgments

We are pleased to acknowledge Spencer Cheng, who initiated this work and formulated the task definition; Helene Wong, who was the first team leader from Nortel side and suggested the title “Kernel Verification”; Lynn Marshall, who supervises and helps us; Marc Granic, Henri Balter, Brian Brummund and Victor Ivannikov, who provide senior management support; Richard Krol and John Shannon, who sponsored initial phases of KVEST development; and KVEST developers and users, who made a lot of contributions in clarification of KVEST approach and helped achieving goals that seemed absolutely non-realistic five years ago.

References

1. B. Algayres, Y. Lejeune, G. Hugonnet, and F. Hantz. The AVALON project: A VALidation Environment For SDL/MSC Descriptions. In *6th SDL Forum*, Darmstadt, 1993.
2. I. Burdonov, V. Ivannikov, A. Kossatchev, G. Kopytov, S. Kuznetsov. The CLOS Project: Towards an Object-Oriented Envirinment for Application Development. In *Next Generation Information System Technology, Lecture Notes in Computer Science*, Vol. 504, Springer Verlag, 1991, pp. 422-427.
3. I. Burdonov, A. Kossatchev, A. Petrenko, S. Cheng, H. Wong. Formal Specification and Verification of SOS Kernel, *BNR/Nortel Design Forum*, June 1996.
4. J. Derrick and E. Boiten. Testing Refining Test. *Lecture Notes in Computer Science*, 1493, pp. 265-283.
5. R.Kramer. iContract - The Java Design by Contract Tool. *Fourth conference on OO technology and systems (COOTS)*, 1998.
6. B. Liskov, J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, McGraw-Hill Book Company, 1986.
7. L. Murray, D. Carrington, I. MacColl, J. McDonald, P. Strooper. Formal Derivation of Finite State Machines for Class Testing. In *Lecture Notes in Computer Science*, 1493, pp. 42-59.
8. D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering*, Vol. 24, No. 3, pp. 161-173.
9. A. K. Petrenko. Test specification based on trace description. *Software and Programming, (translated from “Programmirovanie”)*, No. 1, Jan.-Feb. 1992, pp. 26-31.
10. A. K. Petrenko. Methods of debugging and monitoring of parallel programs. *Software and Programming*, No. 3, 1994.
11. The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall, 1992.
12. The RAISE Language Group. *The RAISE Development Method*. Prentice Hall, 1995.

Internet resources.

13. http://www.fme-nl.org/fmadb088.html
14. http://www.kvatro.no/products/itex/itex.htm
15. http://www.sun.com/960201/cover/language.html