Formal specifications in reverse engineering and software verification

Proceedings of the Russian Academy of Sciences Institute for System Programming. N 1. 1999.

Formal specifications in reverse engineering and software verification

I.B.Bourdonov, A.V.Demakov, A.S.Kossatchev, A.V.Maksimov, A.K.Petrenko

Abstract. *KVEST — Kernel VErification and Specification Technology* — is based on automated test generation from formal specifications. The technology was developed under a contract with *Nortel Networks* and is based on experience gained in academic research [1]. By 2000 the methodology and the toolset have been applied in 6 industrial projects dealing with the verification of large-scale telecommunication software. The first project, named *Kernel Verification* project, gave its name to the methodology and the toolset as a whole. The results of this project are presented in the *Formal Method Europe Application* database [28]. It is one of the largest formal method applications presented in the database. This paper provides a brief description of the approach, a comparison to related research, and prospects for the future work*.

1. Introduction

1.1 Reverse and forward engineering

Software engineering tasks can be conventionally divided into two large groups: reverse engineering and forward engineering. Different researchers and practical software developers pay different degree of attention to these groups, but today no industrial project can ignore problems of each group. Forward engineering is needed for progressive software development, while reverse engineering is needed to support continuity of functionality and such characteristics as reliability, controllability, openness to changes, etc.

In the context of the industrial software design and development, it is important to combine methods and technologies of software analysis and development. An underestimation of the importance of this combination might lead to a situation when some phases of the software life cycle are supported with hypertrophically advanced tools, which among other consequencies causes an increase of software size, while other phases are lacking adequate tools and as a result face insuperable obstacles. An obvious example is the development of programming languages, in particular, object-oriented (OO) languages including compilers and integrated development environments. It led to extremely huge software systems, which are impossible to support, study, and modify without special methods and tools.

In this paper the reverse engineering often precedes the forward engineering. It can be explained by the following two reasons. The first reason is that the authors are closely acquainted with the subject since the reverse engineering was a starting point for works on development of technologies discussed below. The second and maybe the more important one is that the reverse engineering tasks are simpler to formalize in many respects which gives an essential prerequisite to the experimental application of the most advanced software design and development methods and tools on reverse engineering tasks. The simplicity of formalization here results from the fact that the source for reverse engineering — software sources — is fully formalized. In the case of forward engineering, the source is a less material substance: design methods, designer's skills, informal requirement specifications, etc. Thus, despite the fact that the paper objective is to develop a unified approach to solving problems of reverse and forward engineering, further we will discuss aspects of software engineering in the same order: «reverse» first and «forward» next.

1.2 Formal methods in software engineering

It is very difficult to give a precise definition of «formal methods» as they are understood in theoretical programming. One of the reasons is that programs and methods of their compilation and interpretation are undoubtedly formal, so it is easy to declare all methods of software development to be also formal. However, the term «formal methods» denotes something that differenciates the process of writing program texts in programming language from the process of analysis of such texts and the analysis of program behaviour defined by these texts. Such an analysis is close to mathematical research since it uses mathematical notation as well as reasoning and proving techniques that are a common practice in mathematics. In this connection many authors define «formal methods» simply as methods of software development that use mathematical notation and/or mathematical reasoning. We are ready to accept this definition because we do not see any reason to search for a better one.

Apparently formal methods appeared in programming togehter with programming itself. The most famous results of Soviet programming school are in the works of A.A.Markov (Markov algorithm) [24] and A.A.Ljapunov [25] and his school (for example, Janov schemes [26]). Later formal methods were payed much attention in the USSR in the

* The work was partially supported by RFBR grants 96-0101277 and 99-01-00207.

works of scientists from Kiev, Novosibirsk, Leningrad, and Moscow. The most famous and wide spread formal notation is Bacus-Naur form used to describe syntax of formal languages. Also there are Turing machine, *Finite State Machines* (*FSM*) or *Finite Automata* (*FA*), Petri nets, languages for description of communicating processes by C.A.Hoar and R.Milner, etc.

For obvious reasons almost all works on formal methods were focused on forward applications. The following scheme was considered ideal. Functional requirements to a software system are described in a formal specification language. Correctness of specification is determined analitically — specification is verified. Then a program code is generated from the formal specification by some tool. A slightly more realistic scenario enriched the above scheme with the process of step-by-step refinement of the specification. Each step of refinement is performed by a person who guides the refinement process. Special tools control the correspondence between each refinement and the source specification. In both scenarios the result is a software implementation that meets the specified requirements and has no errors.

In 1970s formal specification languages appeared that, on one hand, had much in common with programming languages, and, on the other hand, provided special means that brought them closer to mathematical notation and simplified reasonings on properties of such formal texts. The most famous of such languages are *CSP* [13], *CCS* [14], *VDM* [15], *SDL*[16], *LOTOS* [17].

Despite of this most part of research in formal methods still was, so to speak, «academic». Apparently the main exception are the works on *FSM*, which are widely applied in design and testing of automatic devices, telecommunication and computer hardware. The experience gained in using *FSM* in hardware development was also used in software development, however in a significantly lesser extent than in hardware development.

Very limited results achieved in attempts to apply formal methods in real-life projects caused a sceptical attitude to the ability to gain any profit from using formal methods that would be comparable with the expenses of additional work on development and analysis of formal specifications. However, formal methods and formal specification languages in particular have had a significant success in some areas. On one hand, the success was caused by an appropriate combination of requirements of the problem area and features of the applied formal methods (mostly it is problems of specification of telecommunication protocols; *SDL*, *ESTELLE*, *LOTOS* are languages used in this area). On the other hand, the success was helped by bringing specification languages closer to forms that are common in traditional programming (first of all it is *Vienna Development Method* — *VDM* and it's succsessors — *Z* and *RAISE* languages).

Another factor that helped to promote formal methods into real-life software production is the interest to problems of reverse engineering as a whole and to problems of test automation based on the use of formal specifications. Therefore, coming back to earth from heavens, experts in formal methods discarded the dream to produce error-free programs and decided to use their methods for searching for bugs that are inevitably introduced in software.

The main advantage of using formal methods in the reverse engineering is the ability to strictly define software system interfaces and behaviour. Such an ability allows to fix knowledge about the functionality of components and subsystems, interaction rules, restrictions on input data, temporal characteristics etc. Thus, there is a premise for solving the main problem of the modern reverse engineering. The problem is that now the result of the program analysis (that consistutes reverse engineering in restricted sense) is knowledge of a separate individual. This knowledge is not estranged from the individual and is easily lost by this individual (and group, in which he or she works), and also by the customer of reverse engineering service as soon as this individual changes work. It is known that companies that produce software spend much effort on software documentation. However, just a few of them have sufficient resources and the time to support documentation in the updated condition. This situation each time causes the need in reverse engineering. The real way out of this infinite cycle is in establishment of the so-called «software contracts», which can be considered as a material presentation of knowledge on software functionality.

The software contract describes a syntax and a semantics of system interfaces. As a rule, this term is used in relation to so-called *Application Programming Interface* (*API*). An *API* is an interface which is provided by program entities, for example, procedures, functions, methods of classes etc.

Besides just fixing the software contract, a formal specification allows to systematize the functional testing (what is frequently referred to as testing by a «black box» method). Since formal specifications strictly describe requirements on both the input data and the expected results, functional specifications are sufficient to perform testing of the external behaviour of the system.

Note that without precise specifications such a systematized approach is impossible since there is no information neither about the input domain for the target system, nor about the criteria of evaluation of the obtained results — which of the results should be considered as correct, and which — as wrong. It is one of the reasons for the large part of the research on testing to be devoted to the testing based on source texts. The source texts provide strict description of the implementation structure, therefore they are suitable for extraction of tests (test influences) and for test coverage estimation. However, unlike the functional specifications, it is impossible to make the conclusions about criteria of check of conformance of implementation with its functional requirements and, in particular, about completeness of implementation, based on study of the source texts.

One more circumstance is very important. If the specifications are formal, they can be considered as «machine-readable». Thus, there is a prerequisite for a complete automation of both test generation and analysis of test results.

In the last ten years another direction of formal methods application — «model checking» — has become serious. This approach shows the compromise between an ideal dream of formal system verification and the reality of software development. The essence of this approach consists in creation of a model of the real system and, whenever possible, a complete check of correctness of the given model. If possible, the check is done by analytical methods. If it is not possible, testing of the model is performed. Complexity of the model, as a rule, is chosen such that it is possible to perform exhaustive testing. The weak point of

the given approach are the problems of model creation and the proof that the model is sophisticated enough to properly represent the properties of the real system.

Summarizing the given brief review of positive shifts in the use of formal methods in industrial software development, we shall note that rather a precise division of methodologies and their supporting tools, oriented to academic research and to use in software industry, is now evident. The later differ from the prior not only in the more advanced means for support of software projects, but also in means allowing to make a link between the specifications and the target system. Among such means are compilers of executable subsets of specification languages into programming languages, means for coordination of specificational and implementational entities, means for simplifying the configuration of the target and test systems, in which a part of components is created manually, and the other part is a result of generation from the formal specifications. A variety of such opportunities allows to conclude that some formal methodologies and the sets of their tools are already software products, and a significant expansion of both their functionality and scale of their use should be expected in the nearest future.

2. *KVEST* history and conceptions

2.1 History of *KVEST* Development and Use

In 1994, *Nortel Networks* (Bell-Northern Research and Nortel (Northern Telecom) are the former names of Nortel Networks) proposed ISP RAS to develop a methodology and supporting toolset for automation of conformance testing of *API*. A real-time 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 [3].

In the case of success, *Nortel Networks* was obtaining a possibility to automate conformance testing for the next OS kernel porting and for the new release of the OS. In addition, Nortel was improving 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. Team members had rich experience in operating system design, real-time systems, compiler development, and the use of formal specification in systematic approach for software design and testing [2, 9, 10].

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

During the next half a year, ISP developed the first draft of the specification and test generation methodology and developed a prototype version of specifications and a test oracle generator. The prototype has demonstrated a possibility of use of the 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. The *KVEST* methodology uses a modification of the FDNF (Full Disjunction Normal Form) criterion for the partition of input space, test coverage and test strategy.

During the second year, a product version of the specifications and tools for test generation and execution were completed. From mid-1996 to the beginning of 1997, most of the test suites were produced and the OS kernel was successfully tested. Results of the testing surprised the customer. No one expected to detect several dozens of errors in the very critical part of software that had been used in the field for about ten years.

2.2 Main concepts of *KVEST*

We shall distinguish methodology and technology, a set of the technical decisions and tools supporting *KVEST* methodology. (In Russian a word «methodology» sounds too pretentious, nevertheless we use it as it is shorter, than «set of methods» and is completely adequate to the treatment of the word «methodology» in English language literature). The *KVEST* methodology is focused on establishing software contracts and on various ways of using of the software contracts specifications.

As the basic method of specification the so-called «model-oriented» or «state-oriented» (model based or state based) approach is used. This approach is alternative to the so-called «algebraic» approach, synonyms of which are «axiomatic» and «action-oriented» (action based) approaches [21, 22]. In the model-oriented approach a specification represents a set of functions and data descriptions, on which the given functions operate, that are familiar to programmers. As a rule, each operation (procedure) of the target system corresponds to some function in specification. In case of the data the picture is much more complex. «Visible» implementational data and hidden data are distinguished. The input and output parameters of operations (procedures, methods) are always visible. Such data as global variables, static areas of memory are also declared «visible», if they are included in the software contract (that admittedly contradicts the requirements of competent modular construction of interfaces). Otherwise they are considered «hidden». The visible data are modeled in a manner close to their implementation. The hidden data can be modeled without a direct binding to the implementation. It allows, first, to make the specifications implementationally independent in a greater degree, and secondly, more abstract and, hence, frequently shorter and more clear.

In the model-oriented specifications two kinds of function description are used — explicit and implicit. The first is familiar to all users of conventional programming languages. Such a description represents the description of an algorithm of calculation of function result. The implicit way consists in a description of restrictions on input parameters (pre-condition) and a restriction on the set of input and output parameters (post-condition). Both pre- and post-conditions are predicates, i.e. Boolean functions. The pre-condition is true if and only if the values of input parameters are within the domain of the given function (behaviour of function outside its domain is undefined and consequently is not considered and is not tested). The post-condition is true if and only if the set of values of input and output parameters meets the requirements that specify functionality (purpose) of the given function. Thus, the

post-condition can be considered as formal definition of a criterion of correctness of function result.

Note that the implicit form easily allows to describe not only the specific value that function should calculate, but also the class of allowable values. It is very important when describing real software contracts. We consider two examples. In the first example it is not important whether to define a function implicitly or explicitly. In the second example the explicit form is practically not acceptable.

The first example is a function that performs some integer arithmetic calculations. The formula (one of the possible ones) on which the given calculations are performed is known; we shall designate it as $f(x)$, where x denotes arguments. In such a case the explicit specification of the target function $tf(x)$ will be as follows (X is type of arguments; Y is type of results):

```
tf : X → Y
tf(x) is f(x)
```

An implicit specification could look as follows:

```
post–tf : X × Y → Boolean
post–tf(x, y) is y = f(x) // here x is an argument,
                          // and y is a result of the target
                             function
```

Note that both considered specifications can be implementation-independent, since we do not make any assumptions on what formula the calculations in implementation of the target function are peformed by. The only important fact is that the result of the target function should be the same as the result of calculation by the formula $f(x)$.

The second example: the function that allocates some area of memory should return a descriptor of this area. The implementationally independent specification cannot predict what value this descriptor will have. However, a restriction on such a result is easily specified in the following implicit form: the new descriptor should not coincide with any descriptors of other areas of memory. Usually this requirement together with the requirements specifying type restrictions are sufficient for the specification of such a function.

While revisiting the first example note that if the argument of the target function is a real number, the explicit form will be useless without additional agreements on interpretation of such specifications. When specifying calculations in real numbers it is necessary to additionally specify the accuracy of calculations and the accuracy of the representation of results. For these purposes the implicit form will be more preferable. For example, the specification of the function *tf* could look as follows:

```
post-tf : X × Y → Boolean
post-tf (x, y) is
                abs(y–f(x)) < delta
```

Abandoning the specifications binding to algorithms used in an implementation makes the specifications implementationally independent only formally, not actually. From the good specification, in comparison with an implementation, we expect the more obvious, transparent description of «sense». Here we concern a very delicate subject. It is difficult to give strict definitions allowing to compare two texts so that to specify in which of them the sense is presented «more obviously». Nevertheless, it practically always appears that the «good» specification differs from the implementation in that more abstract data structures are used in the description of interfaces. In turn, it is possible to say that a data structure is then «more abstract» when for its description we use such mathematical concepts as sets, maps, graphs and their variations in a greater degree. Apparently, it is not meaningful to say that the mathematical notation and techniques of the mathematical descriptions, reasonings, transformations are better than the appropriate means used by programmers in their real practice. The only important fact is that the mathematical notation encourages the software designer to look at the program, its implementation, its behaviour from a new angle. Such a new view of the program is apparently the main reason for the effectiveness of formal methods even where completely automatic generation of programs from specifications and complete analytical verification of programs is impossible.

KVEST uses in full measure this technique of description of software contracts with abstract data structures. It is true for both types of visible data and for the description of hidden data. Thus, describing arithmetic functions that work with integers of different length, *KVEST* defines an integer type of infinite length. This technique allows to reveal many errors in implementations of seemingly simple functions. Actually these functions are not complex only in classical arithmetics of an infinite length. Algorithms become not obvious if arithmetics with integer of finite length or unsigned arithmetics are used. The comparison of calculations in «infinite» and in «finite» arithmetics allows to verify libraries of arithmetic operations in unified manner.

In a much greater degree the increase of abstraction level concerns the description of hidden data models. They correspond to so-called «abstract» variables. The structure of abstract variables can be quite different from that of hidden data. Moreover, the structure of abstract and of hidden variables, as a rule, are essentially different. The only requirement for choosing the set of abstract variable is an opportunity to simulate behaviour of the target system, its functions and, thus, as a minimum, to estimate results obtained from target functions.

2.3 Terms and Basic Notions

Let there be some software system containing a 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 the *API* testing. From now on, we will say that *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 these terms synonymous 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 the 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.

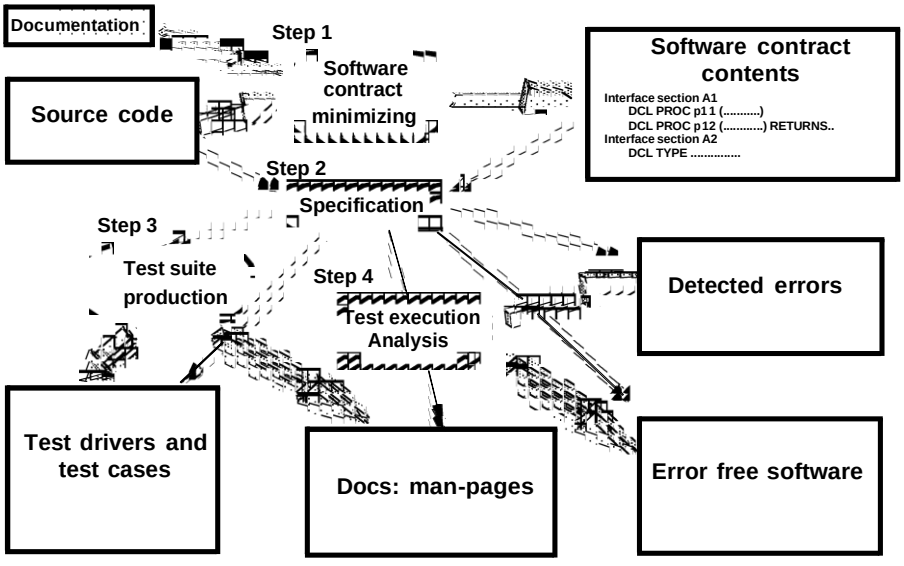

Figure 1. KVEST methodology steps and results.

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

We consider two levels of test drivers. Let us call a *basic driver* 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 parameters and records the 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.

Let us call a *script driver* 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 whether the desired test coverage is complete and if not, continues to generate sets of input parameters and call the basic driver with these sets.

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

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

2.4 The methodology steps

The *KVEST* methodology consists of the following steps (Figure 1.):

- software contract content definition;
- specification development;
- test suite production;
- test execution and test result analysis.

2.4.1 Software Contract Content Definition

The goals of this step are:

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

Following these goals we can minimize 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.

2.4.2 Specification Development

Goals:

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

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

2.4.3 Test Suite Production

Goals:

- to develop the so-called «*manually developed components*» (*MDC*) of test suites;
- to generate test suites.

Most of *MDCs* are convertors between model and implementation data representation, test data structure

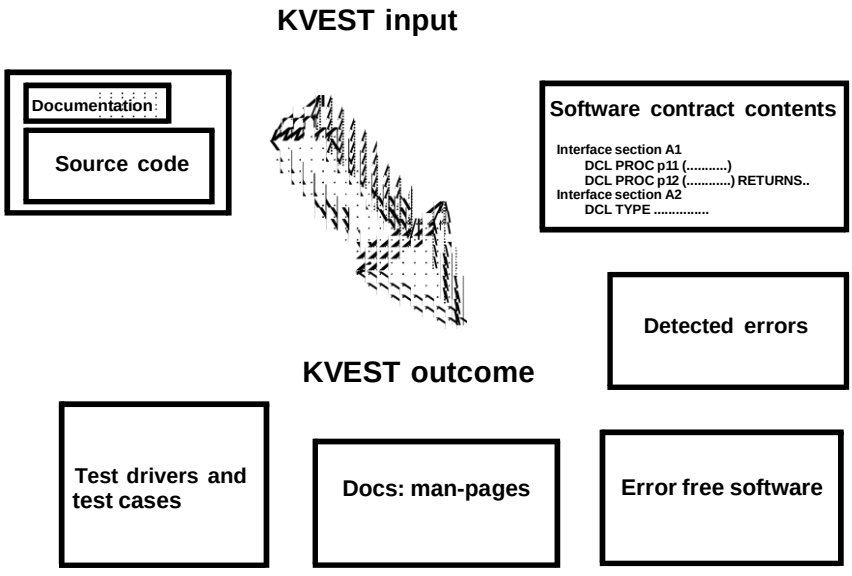

Figure 2. KVEST technology: general scheme.

initiators, and iterators. Based on the *MDCs* and the specifications, *KVEST* generates test suites as a whole. No customization is required after the generation is complete.

2.4.4 Test Execution and Test Result Analysis

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

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

Figure 2 presents a generalized view of the input data and results of using the *KVEST* methodology.

2.4.5 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 an outcome for the target procedure;
- how to estimate the 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 a correct result and False otherwise. So, the generation of test oracles is quite feasible once we have the post-conditions.

A 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 the 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 the input and the properties of the outcome for the target procedures. Both the requirements and the properties are clearly represented in pre- and post-conditions of formal specifications in an implicit form. So, based on the implicit specification, we can successfully solve the problem of test coverage estimating.

There are skeptics who think that full coverage of domains, even including interesting points like a boundary layer, does not guarantee a 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 FDNF representation of the post-condition while taking into account the pre-condition terms too. *KVEST* allows making the partitioning in terms of specification branches and FDNF fully automatically. One of the most difficult problems is the calculation of accessible FDNF disjuncts and removing inaccessible FDNF disjuncts. The problem is solved in *KVEST* by a special technique in pre-condition design.

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

2.5 Test Generation Techniques

2.5.1 API Classification

First, we should consider a classification of *APIs*. The classification determines the choice of one of test generation techniques applicable to a procedure or procedure group interfaces.

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

The classes are organized hierarchically. First class establishes the strongest requirements. Each other 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 values of different 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 the input parameters). The input does not have to be in a literal form. Such procedures can be tested separately.

Example: procedures with the pointer type input parameters.

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

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

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

Example: procedures that provide stack operations and that receive the stack as a parameter.

Kind 5. The procedures cannot be tested separately. A part of the input and output data are hidden and a user does not have a direct access to data.

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

Parallel extension of *API* classes. Theoretically the procedures of all classes should be tested in parallel in case when there is some interaction between procedures belonging to a set. In fact, it is reasonable to conduct parallel testing only for the kind 5 procedures because these procedures share some common resources and collisions are likely in this case.

Example: the procedures for manipulation with mailboxes (send, receive, etc).

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 raises an exception after dividing by zero received as an input parameter because this procedure must not return any return code in case of a zero parameter value.

2.5.2 Script driver scheme. Example of kind 5

The above taxonomy is a good basis for classification of the test generation techniques. Kind 1 allows the 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 a similar structure. The main distinction is the distribution between the automatically generated components and the *MDCs*. Kind 1 script driver is generated fully automatically, kind 2 script driver – almost automatically and so on.

A script driver is a program that is composed and compiled by *KVEST*. A general scheme of script driver is defined by a formal description called *skeleton*. The skeletons are specific for each kind of *API*. Each script driver consists of declarations and a body. The declarations are generated automatically based on the list of procedures under test and their specifications. The declarations include an 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 script driver options 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 the 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 the initialization is finished, the main part of the script driver starts.

Kind 5 script driver implements a general algorithm for traversing an abstract *FSM*. The goal of the algorithm is to pass all states and all possible transitions between the states. *FSM* states here correspond to some classes of the data states. Each transition corresponds to an execution of a procedure under test.

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

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

Instead of a direct specification of an *FSM*, *KVEST* uses its indirect, virtual representation. To describe an *FSM*, a script driver designer should imagine a mental model of the *FSM* and then define a function-observer. The observer calculates on the fly the current state identifier (number) in the abstract *FSM* based on the data accessible to the script driver.

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 had called some target procedures. Now we are going to make the next transition. This elementary cycle of testing consists of the following steps:

- Select an arbitrary procedure from the group.
- Call a set of iterators (data generators for a data type) that prepare a tuple of input parameter values for the target procedure.
- If the iterators have managed to generate a new and correct tuple without violation of pre-conditions, then the script driver calls the corresponding basic driver with this tuple as actual parameters.
- If iterators cannot produce a new correct tuple, then we return to the first step and repeat the attempt for another procedure.
- When the basic driver returns control, the script driver checks the verdict assigned by the basic driver.

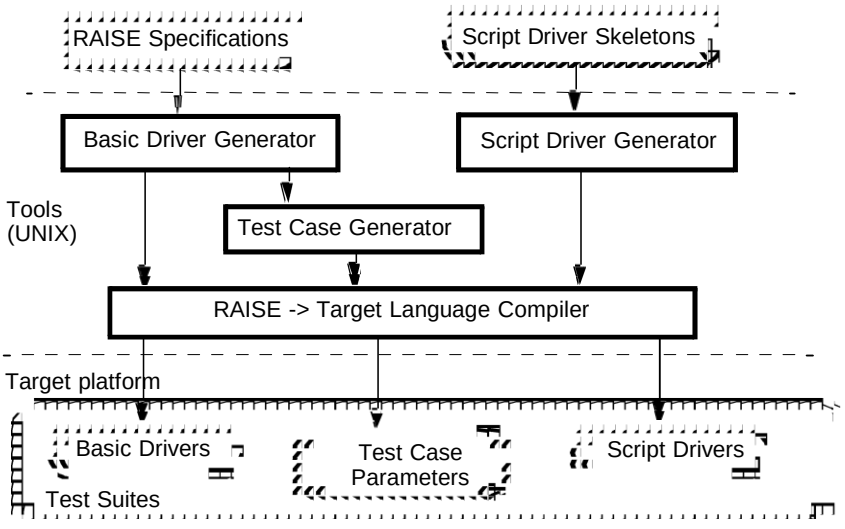

Figure 3. Programming language independent test suite generation scheme.

- If the verdict is False (an error has been detected), the script driver produces the corresponding trace data and finishes.
- If the verdict is True (the elementary test case passed), the script driver calls the observer of the group to calculate the current state number, logs the state number and transition and continues to traverse *FSM*.

2.5.3 Test Suite Composition

Let us go back to the issue of the composition of the *MDCs* and the 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 the 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 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 kind 2 *API*. For other kinds, test designer usually has to add some specific initiators and iterators. In any case, the designer should define an *FSM* state observer for the script drivers of kinds 4 and 5.

Basic drivers invoked by script drivers are generated fully automatically. The only *MDCs* called from basic drivers are data converters. As mentioned above, the converters transform a model data representation into an implementational representation and backward. A model representation is distinguished from the implementational one by the level of abstraction. For example, models may use «infinite» representation of integers, sets, maps, and other data structure suitable for specification. Sometimes the model representation is very similar to the implementational one. In this case, such transformation is done by a standard translation algorithm of 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 components written in the implementation language are data converters. These components are out of the scope of test generators. So, the test generation process produces a full test suite source in *RSL* at first, and then the source is compiled into the implementation language. To port a *KVEST* test suite from one implementation language platform to another, a user should re-write data converters and provide a compiler from *RSL* into the implementation language, as well as a run-time support system with test bed functions. The scheme of test generation is presented in Figure 3.

2.6 Integration of reverse and forward engineering

In 2000 *KVEST* is going to be used in co-verification manner. The team of designers in Ottawa will develop design documentation and source code while another team in Moscow will in parallel develop formal specifications and test suites. The co-verification scheme allows to improve the quality of design documents and to produce test suites before implementation is complete. It is one of the ways to use *KVEST* in the forward engineering process.

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

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

3. State of the art

In this section, we will discuss systems that, on one hand, build the verification process on the formal specifications, and, on the other hand, propose a rather generic technological scheme, not just 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.

3.1 *ITEX – Interactive TTCN Editor and eXecutor*

ITEX [29] 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:

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

Besides, 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.

3.2 ADL/ADL2

This approach [30] is the most similar to the works of our group. From formal specifications, *ADL* generates test oracles and skeletons for building test drivers and documentation. Not very fundamental, but still interesting difference of *ADL* is that instead of one of the popular specification languages it uses 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. It is documented in the Kernel Verification project report [20]. A similar kind of extension was also proposed by Barbara Liskov [6]. There are ideas on extending *Java* and other object-oriented languages aimed at developing software in «Design-by-Contract» fashion [5, 27, 31]. However, despite the obvious advantages of a better acceptance of such languages in the software engineering community, the common concept is still far away, to say nothing about the common notation.

The difference of the *KVEST* results compared to *ADL* results can be explained by the difference in the range of *API* classes for which this or another methodology can provide means for specification and automatic test generation. *ADL* provides adequate tools for test generation automation only for procedures which parameters allow independent enumeration while this set of procedures allows testing procedures one by one. In the *KVEST* classification, these are the *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 a parallel mode, e.g., «lock — unlock», or «send — receive», are left out. Besides, *ADL* does not recognize the first kind of *API* that permits an automatic generation of the complete test suite including test case parameters and test oracles.

An interesting moment in *ADL* is its capability to generate natural language documentation. It is important to note that the same mechanism is used for both the documentation of the target system and the documentation of the test suites. It seems that *ADL* authors made a conscious choice of not using any of the technologies from the *NLG* (*Natural Language Generation*) field. It is easy to explain in a pragmatic sense, however, it does not mean that modern natural language generation methods can not 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.

The significant advancement of *ADL2* compared to *KVEST* is its capability of specification and testing of OO classes. This *KVEST* shortcoming can be explained by *RSL* weakness. Extending *KVEST* to OO software verification is the task for 2000.

3.3 Using Test Oracles Generated from Program Documentation

This work [8] is in a research phase and as such it is not a technology ready to use in industry. The main interest in this research is the analysis of factors that, in authors' opinion, prevent the wide use of formal specifications during automatic testing of industrial software. The authors formulate five main problems the common solution of which, in their opinion, is impossible within the current state of the art. These five problems have a lot in common with a group of characteristics that were the basis of *API* classification in *KVEST*. Thus, D. Peters and D. Parnas and us arrived to the common understanding that those are the key problems in the task of test automation based on formal specifications. *KVEST* continued research in this direction and proposed a technological scheme for a partial automation of test suite development for all kinds of *APIs*.

3.4 Formal Derivation of Finite State Machines for Class Testing

This work [7] is also a research. At the same time, this work is interesting in a 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 build test suites to verify the conformance of the implementation to the specification using formal specifications of the methods for some class. As a test coverage criterion, a union of two criteria is used: to cover all equivalency classes that represent areas obtained as a result of partition analysis, and then, to check the results on or near 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 some tools. Still, all the steps are described in a very systematic way and can be boiled down to various specification transformations.

The partition and boundary analysis is done manually according to the methodology proposed by the authors. In a similar way, they build a specification of oracles. Oracles, once compiled into *C++*, call target procedures and verify the conformance of the results to 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 a Finite State Machine description that represents an abstraction of 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 a mostly manual activity, which limits its applicability to the industrial software.

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

4. Conclusion and further work

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

However, the current state of *KVEST* and the state of the art as a whole dictate the necessity of an intensive development of the approaches, methodologies and supporting tools. It is possible to formulate the following important problems to be solved:

- The users of programming languages are not able to use specification languages and formal methods with sufficient skill, it is the main factor constraining wider use of formal methods;
- Methodologies, technologies, CASE systems supporting software development, as a rule, are aimed at the development and analysis of implementation structure (architecture). This approach makes a consideration of software functionality itself difficult. There are still no methodologies that successfully combine advantages of both structural and functional approaches.
- Static (purely formal) methods of program analysis provide exhaustive decisions of the problems, but are aplicable only for fragments of real systems. Dynamic methods such as modeling or testing are based on some heuristics and therefore cannot form the basis for an exhaustive analysis. In this connection, there is the problem of integration of static and dynamic methods, to take advantages of each of them.

We see the following ways of solving these problems:

- **Rapproachement of specification and programming languages.**

Keeping in mind that it is impossible to force a programmer to study not only a new specification language, but just a new programming language, it is easy to conclude that it is necessary to make a smooth transition from a programming language to a specification language. The attempts to pull these languages together were made for a long time. Examples of such languages are *CLU* extension [18], *Alphard* [19], *Eiffel* [27, 31], *iContract* [5], *SDL* [16], *SPP* [20].

Some languages, for example *Larch* [23], offer to make this rapproachement purely at the expense of a simplification of the specification part while simultaneously expanding the means for displaying of a formal model in the implementation language. Other languages, for example *SDL*, are using the specifities of the problem area, borrow features of programming languages and thus allow to generate executable code directly from the specifications.

There is quite a successful experience of a counter movement. For example, the authors of *ADL* slightly expand *C*, *C++*, *Java*, and *IDL*. As a result, there are no problems for user of the programming language at least in reading specifications.

KVEST in its prospective development considers the last of the listed approaches as the main one. The version of the system currently under development offers means of *C++*, probably enriched by «syntactic sugar» simple in implementation and in understanding, as a specification tool.

Note that the concept of such updating requires a thorough study. A consequence of «simple decisions» is easily illustrated by the example of *ADL*. Absence of methods and appropriate means for increasing the level of abstraction do not allow *ADL* users to create specifications of higher levels of abstraction and in a sufficient degree to automate test generation for groups of procedures or classes.

- **Synthesis of program analysis and program development methods aimed at the description of functionality and implementation structure.**

Some aspects of software behaviour poorly fit in the framework of *API* consideration. As an example there are stacks of telecommunication protocols and distributed systems. Such software is well described in the form of executable models, constructed on the basis of *FSM*, Petri nets, special kinds of automata, for example, *CCS* [14]. Common disadvantage of such approaches are the difficulties with setting invariants for distributed events. In particular, this problem showed up in attempt to detect a potential undesirable feature interaction. One of the ways of structural and functional methods integration are *FSM* extended with description of restrictions on transitions [4]. This approach is close to the technique of describing and testing of procedures groups proposed in *KVEST*. Thus, *KVEST* can be extended by appropriate means for describing protocols, distributed systems and other kinds of software requiring combination of the structural and functional specifications.

- **Integration of static and dynamic methods of forward and reverse engineering.**

Among potentially possible applications of the synthesis of static and dynamic methods of development/analysis, we

recognize means for controlling level of abstraction and generators of models for static and/or dynamic analysis of program behaviour.

As a means for increasing level of abstraction, methodical and tool support for «upwarding» and «downwarding» together with means for transformation of explicit specifications into implicit ones (in the form of post-conditions) is developed within *KVEST*. Until the present time these works were carried out completely manually. Now some of the planned «transformers» are being developed.

The models under consideration are generalized descriptions of scenarios of the target system usage. Such scenarios can be used both for static analysis (this is what the idea of «model checking» means) and for generation of test sequences. In *KVEST*, *FSM* in the 5-th kind of test drivers played a role of such models. The fact that such *FSM* is developed manually, besides problems connected with cost and terms of its development, raises a question of correspondence between *FSM* and procedure specifications, of the relation between test coverage criteria defined on the basis the specifications and those defined on *FSM* itself. There are the works on partial automation of such *FSM* construction. Unfortunately, problems of *FSM* extraction from the specifications of functions with side effect are not considered in them. *KVEST* makes an attempt to solve this problem using specifications of hidden (abstract) states of target systems.

References

I. Bourdonov, A.Kossatchev, A.Petrenko, and D.Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. In Proceedings of World Congress of FM'99 — Formal Methods, Lecture Notes in Computer science, volume 1708, Springer Verlag, 1999, pp.608-621.

2. I.Burdonov, V.Ivannikov, A.Kossatchev, G.Kopytov, S.Kuznetsov. The CLOS Project: Towards an Object-Oriented Environment for Application Development. — In Next Generation Information System Technology, Lecture Notes in Computer Science, volume 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. Pansy Au and Joanne M. Atlee. Evaluation of a State-Based Model of Feature Interactions. // Proceedings of the Fourth International Workshop on Feature Interactions in Telecommunications Software Systems, June 1997, pp. 153–167.
5. R.Kramer. iContract — The Java Design by Contract Tool. // 4th 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. // Lecture Notes in Computer Science, volume 1493, pp. 42–59.
8. D.Peters, D.Parnas. Using Test Oracles Generated from Program Documentation. // IEEE Transactions on Software Engineering, 1998, Vol. 24, N. 3, pp.161–173.
9. A.K.Petrenko. Test specification based on trace description. // Software and Programming, New York (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, N. 3, 1994.
11. The RAISE Language Group. The RAISE Specification Language. — Prentice Hall Europe, 1992.
12. The RAISE Language Group. The RAISE Development Method. — Prentice Hall Europe, 1995.
13. A.R.Hoare. Communicating Sequential Processes. — Prentice Hall, 1985.
14. R.Milner. Communication and Concurrency. — Prentice Hall, 1989.
15. D. Bjorner et al eds. The Vienna Development Method: The Meta-Language. — Lecture Notes in Computer Science, Vol. 61, Springer Verlag, 1978.
16. Specification and Design Language. ITU-T recommendation Z100.
17. P.H.J. van Eijk et al eds. The Formal Description Technique LOTOS. — North Holland, 1989.
18. Barbara Liskov et al. CLU Reference Manual. — Lecture Notes in Computer Science, volume 114, Springer Verlag, 1981.
19. Mary Shaw. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. // Communications of the ACM, Vol. 20, N. 8 (August 1977), pp. 553–563.
20. SPP — specification language description. — KV project report, Nortel (Northern Telecom), May 1995.
21. C.A.R.Hoare. An axiomatic basis for programming. // Communications of the ACM, Vol. 12, N. 10 (October 1969), pp. 576–583.
22. C.A.R.Hoare. Proof of correctness of data representations. // Acta Informatica, 1(4): 271–281, 1972.
23. J. Guttag et al. The Larch Family of Specification Languages. // IEEE Software, Vol. 2, N. 5, pp. 24–36 (September 1985).
24. А.А.Марков. Теория алгорифмов. // Москва, Изд-во АН СССР, 1954.
25. А.А.Ляпунов. О логических схемах программ. Проблемы кибернетики, вып. 1, Москва, 1958.
26. Ю.И.Янов. О логических схемах алгоритмов. Проблемы кибернетики, вып. 1, Москва, 1958.

Internet resources

27. http://www.eiffel.com/doc/manuals/language/intro/
28. http://www.fme-nl.org/fmadb088.html
29. http://www.kvatro.no/products/itex/itex.htm
30. http://adl.xopen.org
31. http://www.elj.com/eiffel/intro/