Experiences in using testing tools and technology in real-life applications[1]

A.K.Petrenko, I.B.Bourdonov, A.S.Kossatchev, V.V.Kuliamin
Institute for System Programming of Russian Academy of Sciences (ISPRAS)
{ petrenko, igor, kos,kuliamin } @ ispras.ru
http://www.ispras.ru/~RedVerst/

Abstract
The Institute for System Programming of the Russian Academy of Sciences (ISPRAS) in cooperation with Nortel Networks has gained a unique experience in the practical application of formal specifications of Application Program Interface (API) functionality and automated specification based test generation. This approach and supported tools have been used in several industry projects and presented in a several world conferences and workshops. This paper presents an outline of the this approach and issues gained from its application during the last six years. The newly developed UniTesK concept is presented as an alternative method for Object-Oriented software development and test generation. The specification based testing provides new opportunities in shorting total software life cycle length, decreasing maintenance and regression testing cost, and increasing software reliability. Furthermore the paper considers the problems of specification based testing introduction in industry and prospective features that facilitate the introduction.

1 Introduction

1.1 Technical, management, and human problems

Verification and validation are necessary components of software development process. Both ones need wide range of specifications from general requirements specification (maybe informal or semiformal descriptions) to detail specification of implementation units. The only specification use allows to conduct full and qualitative checking software product and analyzing development processes progress and quality. However specification based testing, corresponding techniques and tools still are not widely used in comparison with techniques and tools for implementation based ("white box") testing. There are three sets of reasons and, correspondingly, explanations of the current specification based testing situation: technical, management and human related ones. Unfortunately, in the issue there is not any "philosopher's stone" that solves all problems. In industry context we have to find a common, compromising solution. From this point of view ISPRAS's methods and tools are probably interesting not only as themselves but as example of compromise between the advanced techniques and industry reality.

Technical context. The key technical problems are as follows:

- How to specify requirements, behavior, or functionality of target software
- How to design to meet the requirements
- How to verify

Lets consider first item "how to specify". There are several specification approaches and languages. Selection and adaptation of the features is undoubtedly important issue. However, an answer here depends on:

- Goal of the specification (for documenting, verification, understanding and so on)
- Who will use the specification (personnel background, preferences)

[1] The work was partially supported by RFBR grant 99-01-00207.

- Timeframes (does one have time for studying new notations and technique, how long time period of the technique use)
- Quality requirements (strong or moderate ones)

It is example of close relation technical and non-technical facets of specification based testing in industry context.

The management issues themselves have an influence on and depend on prioritization in software development processes goals. The most significant goals are as follows:

- To short time of development and delivery
- To decrease effort and cost of development
- To build comfortable, creative, productive atmosphere in a development team
- To decrease effort and cost of maintenance
- To guarantee high quality (to obtain good reputation)

The only last item directs introduction of formal specifications and specification based techniques. But the goals should be considered in the context of others ones, so in fact always a manager finds a compromise.

Human facet of any innovation introduction is very important. Here we face usual situation, there are some evident advantages of specification based testing, however the introduction raises new problems for everybody and nobody likes any extra problems. For example, the designers and testers have to study formal specification languages, they have to install and additional tools, they face problem of conversion of specification notions and formats into programming language ones. These problems cause the resistance of potential users to introduction of formal specification and specification based testing. To overcome the resistance we could facilitate studying and introduction of the techniques and integrate the techniques and supported tools with usual software development tools and environment. It is, again, an example of the close relations of between technical and non-technical problems in the area.

Finalizing the section, lets resume once more: to introduce specification based testing in real life we should solve three sets of problems declared in the above heading together.

The issue is confirmed by ISPRAS experience in academic research and industry programs. Because Russian (Soviet) scheme of cooperation between academy and industry is fully different from western one, it is pertinent to briefly describe the ISPRAS's background summered below.

1.2 ISPRAS history, background, partners

The Institute for System Programming of the Russian Academy of Sciences (ISPRAS) experience was gained in academic research and Soviet space programs [ISPRAS]. ISPRAS took participation in development of software for soviet mainframes, mission control centre software, space shuttle on-board computer operating system and real-time programming language. ISPRAS has long-term partnership relations with RFBR (Russian Foundation for Basic Research), Nortel Networks, Microsoft Research, ATS APS (Advanced Technical Services APS), IFAD (Denmark), GMD (Germany), INRIA (France) and others. ISPRAS has concentrated the verification and specification activities in

RedVerst[2] group [RedVerst]. This lecture presents RedVerst group's research and industry results.
RedVerst has developed Kernel VErification and Specification Technology (KVEST) [KVEST, KVEST2000]. The technology is based on automated test generation from formal specifications. To date, the methodology and toolset have been applied to over six industrial projects dealing with the verification of large-scale system and telecommunications software. The first project, Kernel Verification, gave its name to the methodology and the toolset as a whole – KV and then KVEST technology. The results of this project are presented in the Formal Method Europe Applications database [FME]. It is one of the largest applications of formal methods presented in the database.

1.3 Scope of the consideration

First we present position and keystones of the RedVerst approach. Next sections introduce terminology details and brief state of the art. The main sections of the paper present specification and test generation approach themselves. We begin from KVEST technology and then present UniTesK technology.
The given techniques are applicable to wide area of software and have close relations with other approaches to verification and validation of software. However, to concrete focus of our consideration following issues will be (almost) omitted below

- White box testing
- Analytical verification
- Applicability and particularity of the approach in case of specific kinds of software like compilers, real-time systems, databases, protocols, distributed systems and so on.

2 Position

We are considering both methodological and real-life industry problems. So we split the views into 2 parts.
Methodological view:

- Software development improvement needs **joint consideration** of both the software development techniques and software development processes issues and both software design (forward engineering) and software verification and validation activities.
- Main paradigm of RedVerst approach is the thesis about the consideration of software and SDP as **models**. It is acceptable and, moreover, is fruitful to take into consideration a variety of models of an entity.
- The **multiplicity** of the models, their joint consideration and comparison is the only way to know how to establish that the target task is solved correctly and completely.

Industry view. To obtain industry strength the verification and validation technology should solve following problems:

[2] RedVerst stands for **Re**search and **d**evelopment for **Ver**ification, **s**pecification and **t**esting.

- **Scalability** of the techniques
- **Through** computer aided process
- **Both reverse- and forward-**engineering processes should be support
- Always, in case of the legacy reverse-engineering and re-design, and in case of newly developed software, there are some (re-usable) artifacts, which can be, and should be used. Producing re-usable artifacts could be considered as a surplus with relation to main workflow. However, the surplus is very important, so, aims of the process are simplifying accumulation and use of these surpluses for **re-use**. We like to push the developer to produce and keep these surpluses for future use.
- Among the different kinds of legacy there are not only target software components, but also components of the specifications and test suites, so we will jointly consider r**e-use of specifications and test suites**.
- Although reducing time length of development of each release is very valuable, we should not overlook the **long-term prospect**. It is important to decrease terms and efforts, needed for the series of releases. It means, that realizing «rapid development» it is not necessary to save of each of the release, excessively. The reasonable additional effort will be paid back on the following releases.
- Any solution of a problem, both traditional and super-modern, should be **oriented to the user mentality**: the complexity should be compensated by efficiency, the novelty - saving of time and effort, additional efforts - aesthetic pleasure, which gains «well-clean» result.

3 Terminology and background

The section is intended clarifying some terms related with specification and testing.

3.1 Testing

Verification vs. testing. *Verification* is "the process of evaluating a system or component to determine whether or not the products of a given development phase satisfy the condition imposed at the start of that phase" [ESI-verification]. So, verification makes an accent on partitioning development process onto steps, stages, or phases as opposed to *validation* where one is interested in evaluating the target system only at the end of the development process. The validation is closer to acceptance testing and system testing, whereas verification is conducted both during development and when target system is completed too. So verification is closer to unit and integration testing. Verification approach does not restrict means for evaluating a system. *Testing* is one of means for verification that include executing part or a whole target system. Both testing and verification need requirements description to be able to compare actual and expected (or right) behavior. Formal specification is the means for rigorous description of the requirements.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model [Clarke&Wing]. Because testing has to restrict number of tests, test designer has to follow a finite model too. The most researches related with model-checking consider analytical verification approach (without model execution). In fact model-checking consists of two sub-tasks: how to build an adequate finite model and how to verify the model. Testing could use the same techniques to building the finite models but compare the model behavior with target system behavior by means of executing target system. So, sometimes testing and model-checking could follow the same way.

Black box and white box testing. There are two well-known terms: *black box* and *white box testing.* Black box approach considers a system or a component whose inputs, outputs, and general function are known but whose contents or implementation are unknown or irrelevant [ESI-black box]. In fact there is a wide spectrum of testing approaches that use different combinations of black and white box testing. Below we consider one of the combinations open-state testing (as opposed to hidden-state one). The *open-state testing* considers a system as a black box whose internal state (at least informative components) is known but whose implementation (algorithm) is unknown or irrelevant. The open-state approach opens good prospects for quality testing improving.

Test coverage, test coverage criteria, test coverage metrics. There are well known test coverage criteria based on the source code structure. They include criteria based on control flow and data flow structures. The same approach could be applied to formal specification. The test coverage criteria could consider checking test situations related with execution of a specification branch or a combination of logical terms constituted the branch conditions. Because we consider the specification testing below we mean only specification based test coverage criteria.

Testware components: test oracles, test sequences, test scripts, test harness, test bed, test suite, specification suite, verification suite. There is well known term "test case". We are trying to do not use the term because it is poly-semantic one. The following components of the "testing world" are considered:

- *test oracles* (KVEST's basic driver) is a program unit (e.g. function, procedure, object) that checks pre-condition; invokes target operation (operation under test), checks post-condition, assigns a verdict (passed or fault), and generate trace and other data for test coverage and test execution analysis;
- *test* ***scenario****s* (KVEST's script driver) is a program unit (e.g. module or main program) that invokes (possibly indirectly via test oracles) target operations; test scenario checks test oracle verdicts and maybe some additional assertions and assigns a final verdict; in common case, scenario could represent a parallel combination of test oracle invocations;
- test scripts (KVEST's test plan) is a test job description for batch test execution, usually includes a set of test scenarios with definite parameters for each test run;
- test harness is an executable testing program that represents whole test environment for system under test; test harness consists of test bed and test suite in an executable representation;
- *test bed* is the invariant part of the test harness (used for a range of systems under test), function of test bed is run-time support, process and resource management, communication with remote testing tools and databases;

- *test suite* is a target system specific (non-invariant) part of test harness, consists of the test oracles, test scenarios, data iteraters, filters, and so on;
- *specification suite* is a set of specifications (pre- and post-conditions of operations/functions/methods, data type invariants, axioms, specification units like modules or classes, etc.) and auxiliary components like model state, mapping functions for up- and down-warding;
- *verification suite* is a composition of specification and test suites, and test scenarios; the verification suite is static environment of the target system under consideration.

3.2 Specification

There are a few kinds of classification of specification approaches like model-oriented vs. property-oriented and state-based vs. action-based. We are suggesting following classification: *executable, algebraic (in fact, co-algebraic), use cases or scenarios, and constraints*. Below you find short descriptions of the approaches and examples of methodologies and tools used the approaches.
Executable specifications, executable models. The approach supposes developing a prototype system to demonstrate feasibility and functionality of further implementation. The examples of the approach are SDL, VDMTools, explicit function definitions in RAISE.
Algebraic specifications assumes a description of properties of some operations' compositions (serial, parallel, random, or some combinations). Usually the approach is tightly related with axiomatic approach. SDL use the approach to specify data types (ASN); RAISE provides quite powerful mechanism of axiomatic specification.
Use case/Scenario based specification approach suggests considering the scenarios of use instead of properties of the implementation. The approach is developed and propagated by OMG [OMG], Rational Corporation [RUP], SDL community uses MSC [MSC] notation for scenario description. The informative review of the scenario-based techniques is presented in [Ryser].
Constraint specification assumes the description of data type invariants and pre- and post conditions for each operation (function, procedure). There are specific techniques for OO classes and objects specification. The constraint specification approach is followed by classic VDM [Bjorner, Jones], Design-by-contract in Eiffel [Eiffel], implicit function definition style in RAISE, iContract [iContract], ADL and ADL2 [ADL].

The most kinds of specification notations are usually non convenient for software engineers and non-expert customers, it is common drawback of formal specification techniques. Use case and scenario like specifications maybe the most suitable for non-professional society, however the kinds of specification do not contain some knowledge needed for documenting and exhaustive testing.
Each above approach has some advantages and drawbacks. For example, algebraic specifications are very suitable for test scenario generation include case of concurrent and distributed software, however the approach provides test oracle[3] generation techniques only

[3] *Test oracle* is a program or program component automatically compared actual behavior or result of a target operation against its specification.

for pure function, so real-life software designers face some troubles in attempt specifying their software using only algebraic approach. Notice, the algebraic specifications provide very poor base for partition analysis[4]. Maybe the most drawback of the algebraic specifications is non-scalability of the approach. Such specifications for toy example are very short and attractive, however as the size increases, the understandability of the algebraic specification drastically drops.
Lets anticipate a little and clarify RedVerst position. The RedVerst approach follows last direction. We consider constraint specification as main part of specification suite. Other kinds of specification could be used too, but they are additional and optional features for the most specification and verification activities in RedVerst processes. The reason of the choice is as follows. The constraints specification (pre- and post-conditions and data type invariants) allows automatic test oracle generation and conduct partition of domain areas. It is well-grounded base for most reverse-engineering activities and, partially, forward-engineering activities including, test scenario design, test coverage criteria establishment, test coverage estimation, documentation, representation of reverse-engineering results, redesign, and improving design.

4 State of the Art

Below we are considering the notations and tools related with the specification based testing.
Eiffel [Eiffel]: Eiffel is an OO programming language with facilities for assertion specification and support of debugging. Eiffel proposes a methodology **"Design-by-contract"** supported step-by-step process for description of interface functionality and implementation of the functionality. There are interesting results in building re-use Eiffel libraries. Eiffel has wide user community, however, it is known in academic area rather than in industry. We know no specific tool for test generation based on Eiffel specifications that is applicable for industry software.
iContract [iContract]: iContract is an annotated extension of the Java language, which allows users to define pre-, post-conditions and constraints related to the data structures. With the help of a pre-processor, instrumented code is produced, which is used to automatically track the preservation of these conditions throughout the execution of the code. This extension lacks the methodology and dedicated tools for the design of the tests, and is not integrated with software development environments.
Larch [Larch]: Larch is a very well thought out system consisting of a shared specification language and a language binding the specification to the target implementation. The system lacks test suite design tools. Also, it has the same drawback as KVEST, having a formal specification language at the centre of the technique, thus requiring expertise in the specification language as well as the programming language.
VDM-SL, VDM++ [VDM_SL], [VDM++]: Meta-IV (language of classic VDM), VDM-SL, and VDM++ form a widely used family of specification languages, well integrated with

[4] *Partition analysis* divides domain area into a set of equivalence classes or sub-domains. Sub-domains' enumeration and geometry is very informative source for test generation and test coverage estimation.

such popular programming languages as C/C++ and Java. However, no test suite design tools have been provided for industry use yet.

ADL/ADL2 [ADL]: The ADL/ADL2 family of notations is based on conventional programming languages (C, C++, Java, and IDL). The notations offer capabilities to create specifications suitable for the generation of test suites and user documentation. A shortcoming of ADL is in the weakness of the test suite generation tools of operation group testing.

SDL [SDL]**, MSC** [MSC]**, TTCN** [TTCN]: Specification languages formally supported by international standards bodies and targeted to the telecommunications include SDL, MSC and TTCN. The first two languages are intended for the specification of real-time, reactive systems such as those encountered in the telecommunications industry. TTCN is intended for specification of test suites in similar areas. These languages are widely used in their niche areas of application, and they have well-integrated support for specification, implementation and testing. On the downside, aside from their narrow specialization, these techniques require at least two different languages as well as a programming language. This drawback is partially compensated for by the well-defined standards, and by the widespread use of these languages in their niche applications.

There are two valuable researches related with specification based testing. D.Peters and D.Parnas suggested a tabular notation for assertion description and a technique for test oracle generation. They investigated troubles faced during introduction of the approach into real-life software verification processes. L.Murray et al. [Murray] consider the possibility of extracting FSM specification from pre- and post-conditions. Such FSM could be used for test scenario generation for OO class testing. The approach result is very promised, but we know nothing on product tools supported the idea.

There are some researches on application of algebraic like specification for test generation. The most known classical work is [Doong&Frankl]. One of the most recent results was published in [Antoy&Hamlet]. It seems the results still are academic- rather than industry-oriented ones. However, notice, S.Antoy and D.Hamlet have begun to consider in the context of algebraic approach the mapping problem. The problem is considering formal description of the relation between specification (model) and implementation. The mapping problem is usually solved in model-oriented specifications and very important for specification based testing. So, these authors have suggested a promised technique for deployment algebraic specifications in a practical use.

5 RedVerst techniques, experience, and results

5.1 KV: KVEST prototype

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 *Application Program Interfaces* (*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. In 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 Networks was improving its product software structure as a whole, since during software contract definition ISPRAS promised to establish minimal and orthogonal set of interfaces for the OS kernel.

The Kernel Verification project gave name to KV and then KVEST methodology. Duration of prototype phase of the project was about 1 year. The prototype KV approach suggested

- Specification extension of Nortel's in-house programming language – SPP;
- Techniques for implicit specification writing and test oracles generation;
- Specification of the most OS kernel interface procedures.
- A part of test scenarios that were manually produced

5.2 KVEST methodology

During next 1.5 year product version of KV methodology and tools were developed. Later the methodology and toolset have been named KVEST – Kernel VErification and Specification Technology. To cover all OS kernel subsystems KVEST team designed 6 kinds of test scenario schemes. The simplest schemes were intended for separate testing of pure functions (without side effect) and allowed fully automatic test generation. The most complex schemes allowed testing parallel execution of resource managers, like memory and process management, messaging subsystems, exception handling, and so on. The schemes classification is described in [KVEST].

RAISE specification language (RSL) [RAISE-language] was used for specification. KVEST included techniques for automatic and semi-automatic test generation, automatic test execution and test result analysis and reporting. The techniques were oriented on use in real-life processes, so re-generation and repeated re-run of tests were developed in fully automatic fashion.

Five patent applications have been filed based on the KVEST development experience. The most valuable issues covered by the patents are as follows:

- Enhanced test generation technique that allows to exclude from consideration the inaccessible and redundant test situations.
- Programming language independent technology scheme for test generation.
- Automatic integration of generated and manually developed component of test suites for semi-automatic test generation. The technique allows to exclude any manual customization during (after) repeated re-generations of test suites. The feature is valuable for both test design and regression testing period.
- Test generation for parallel using procedures based only serial behavior specification.

Besides, an umbrella patent application covers total technology scheme of KVEST specification based process.

5.3 Applications

As we mentioned above KVEST is intended for specification and testing of API. However there are specific requirements in case of testing a specific software and specification and test have to solve the problems. Up to now KVEST users have gained successful experience in verification of the following kinds of software.

- Operating system kernel and utilities
- Fast queuing system for multiprocessor systems and for ATM framework
- Telecommunication protocols as a whole and some protocol implementation subsystems like parsers.

5.4 Software verification processes

The KVEST are applied in two kinds of software verification processes. First one is “Legacy reverse-engineering and improving process” and second one is “Regression testing process”.
“Legacy” process has been described in details in [KVEST]. It consists of the following phases:

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

The same work provides statistics for effort distribution and results (detecting errors) of the phases. In general, specification and test design need the same time (specification needs more experience). At least 1/3 of the defects are detected during specification phase and about 1/3 are detected by means of test execution. A significant part of errors and other inconsistencies are detected during test result analysis. Evidently, some errors are located in the specifications too, so testing (as by-product) is aimed for both the target software and the specification improving.

5.4.1 Regression testing

The KVEST regression testing process consists of:

- New release reception, determining difference in interface contents
- Specification suite and test suite content fixing
- Test suites re-generation
- Test suites running, comparison of results, error localization and description, recommendation on improving, test result reporting, (optionally) specification fixing if a new functionality appears

Now length of the regression testing cycle is about 2-3 weeks. The rate is quite acceptable for matured software products. Newly developed software requires higher rate. To meet the requirements KVEST has suggested a specific verification process scheme called *“co-verification”*.

5.4.2 Co-verification in forward-engineering processes

The development of formal specifications and the test suites in parallel with the implementation results in the early availability of the test suites before the completion (or even beginning) of the implementation phase. This reduces the cycle by many weeks. Another important benefit of the process is a clear scheme of work “parallelisation” when high quality and fast delivery of the software is a must. When several parallel development streams are started, it is known that an increase of the number of the developers can reduce the effectiveness of the group and may lead to a drop in the software quality. The integration of the resulting work and timely cross-propagation of changes is a daunting task. When a co-verification team is in place, integration of the product comes naturally, as such a team would view the product from an orthogonal point of view. The target of their work is a formal specification of the product, which is more or less independent of the implementation, but ensures the functional consistency of the components.

5.5 Basic research

The RedVerst group conducts three kinds of mutual related activities: academic research, education, and participation in industry projects. All three directions are tightly and mutually related because they need inputs and stimulus from each other. Our basic research directions are as follows:

- FSM extraction from implicit specification
- FSM factorization for test scenario set minimization
- Specification extension of programming languages
- Scenario-based test scenario design

The results of first two topics have been published [Bourdonov]. Two last items are considered below in the section "UniTesK – specification based testing for practitioners".

6 Further prospects. UniTesK – specification based testing for practitioners

The KVEST [KVEST] methodology uses RSL as a specification language and successfully manages the above problems. However, several projects carried out according to this methodology showed that the use of formal specification languages is a serious obstacle for wide application of the developed technique in industrial software production. First, the specification language and programming language often use different semantics and may even use different paradigms (object-oriented and functional, for example), so the special conversion technique must be used for each target language. Second, only developers having special skills and knowledge can efficiently use a specialized specification language.

The solution of these problems is the specification extensions of widely used programming languages. This article presents J@va, a specification extension of Java language, and concerns J@va solutions of problems of specification design and test development. The development of J@va used the KVEST experience and extended it significantly with such features as user-written test scenarios, axioms and algebraic specifications.

The decision to develop a new Java specification extension was based on the fact that existing ones such as ADL and iContract are ill-suited for our goals. They both support oracle generation, but do not support any specification of required test coverage. This fact makes them inapplicable for serious automated test scenario generation. Only ADL permits to write specifications separately from the code specified, but ADL specifications are linked with the corresponding source code in a too rigid way. This constitutes the great obstacle for specification reuse and abstraction level management. A main drawback of ADL and iContract in comparison with KVEST is the lack of support for object state specification and state-oriented testing. These features are necessary for testing of combined behavior of several class methods.

UniTesK stands for **Uni**form **Te**sting and **S**pecification tool**K**it. UniTesK is both approach and technology supported by a set of tools. The keystones of the approach are as follows:

- **UniTesK** provides specification extension of widely used programming languages (now only Java extension “J@va” has been designed and formally described, C++ extension is expected by the mid of 2001).
- **UniTesK** includes facilities for constraint (pre- and post-conditions, class invariants) specification and test design (axioms, equations, test scenarios).
- **UniTesK** tools can be tightly integrated with the most Software Development Environment (SWDE).

6.1 JavaTesK Technology Overview

JavaTesK is projection of UniTesK concept to Java platform. JavaTesK technology process description, like any other software development process (SWDP) description, must include the following components:

- Process Phases
- Process Workflows
- Process Activities
- Process Workers (the roles of process users)
- Process Artifacts

JavaTesK technology can be used as a separate technology process for test development based on formal specifications of existing system (reengineering approach) or can be integrated in forward-engineering. i.e. usualsoftware development process to introduce specification based tests in it (forward engineering approach).

In this overview we shall concentrate only on specific JavaTesK components concerning formal specifications, test scenarios and tests based on them.

6.2 Process Phases

The JavaTesK process has, as many other software development processes, four phases.
Inception. The general goal of this phase is to achieve an agreement among all stakeholders on the global objectives for the project.
If JavaTesK is used as a part of general software development process, the objectives for all JavaTesK-specific activities should be clearly defined.
Elaboration. The overall goal of elaboration phase is to baseline the architecture of the product to provide a stable basis for the bulk of design and implementation in the construction phase. JavaTesK technology demands to supply the baseline architecture with a set of critical use case or test scenarios and specifications, which become the formal expression of the architecture.
Construction. The goal of this phase is to complete the development of the product based upon the architecture elaborated at the previous phase.
JavaTesK specific activity at this phase is the verification suite development, which should be closely integrated with design in forward engineering approach. All the unit tests

developed for this phase should be used immediately after the implementation is completed.

Transition. The focus of the transition phase is to ensure that the product is available for its end users.

When JavaTesK is used as a part of general software development process this phase is mostly target process specific. In the case of JavaTesK reverse-engineering projects the product is the resulting verification suite included both specifications and tests. Contents and nature of the results are essentially different from outcome of usual reverse-engineering activities.

6.3 Process Workflows, Main Activities and Workers

Requirements Elaboration. Used in software forward engineering JavaTesK demands to record the requirements in the form of use case scenarios and formal specifications written in J@va. Scenarios are convenient for capturing complex behavior and simultaneously for contacts with different stakeholders. Specifications are used in this activity to describe clearly the properties and simple functions of entities we meet with in problem domain.

In JavaTesK reengineering project this activity is performed in parallel with Target System Analysis.

Target System Analysis. This activity is specific for JavaTesK reengineering projects. Its purpose is to educe the requirements for the target system, its architecture and design decision made during its development. Unfortunately, critical design decisions as well as requirements often can hardly be educed from target system source code, binaries and documentation. This problem can be solved only by permanent contacts with target system architects and designers or even by inclusion some of them in the project team.

Design. JavaTesK requires using J@va specifications to settle all design decisions. This work not only produces a storehouse of knowledge and design decisions, which are formulated in unambiguous and rigorous form, but also helps greatly to clarify the design details for designers themselves and developers.

In JavaTesK project design of specifications and test scenarios plays an important role. As in general software development to organize specifications, test scenarios, mediators and iteration classes in a proper way may be critical for project success.

Specification design is an iterative activity. At first, the specification designer should determine the functionality to be specified according to the project architecture and design of the target system. In the case when objects of one class or clusters of objects with one dominant represent this functionality in existing system so called *prime specification* can be developed. They consist of proper specifications describing the necessary functionality in implementation terms and mediators linking these specifications with corresponding implementation. Often prime specifications can be generated completely automatically after pointing at the corresponding source code.

At next steps specifications are elaborated to become more abstract and implementation independent. After several steps they should describe only the necessary functionality in a full and precise form, leaving implementation details outside of their scope — they become *complete specifications* of the target system.

One feature of J@va specifications is not intended directly to describe the functionality — it’s coverage metrics. They became important when we want to estimate the effectiveness of testing and often designed according to the structure of implementation and some general hypotheses on possible errors in the target system. Several coverage metrics are extracted automatically from specification structure. The usage of these metrics is valid because in most cases general structure of implementation resemble the structure of corresponding functionality, and so, specifications structure.

JavaTesK technology requires test scenarios to be developed before full specification based test suite can be obtained. While specifications are necessary to describe what is the proper behavior of the target system, test scenarios determine the sequence of target operations to be called to obtain the necessary test coverage. So, the main problem of test scenario design is to organize the “right” scenario of calls with “right” parameters. The use case scenarios recorded during requirements elaboration can greatly help in developing test scenarios, because they define most often used scenarios of calls and important control flow branches.

Another way to design test scenarios used for simple specifications is to organize the systematic traversal of corresponding state-transition graph trying to traverse every transition. Such scenario can be developed with minimal developer intervention. The developer should only specify the way of parameters iteration in consecutive calls. J@va language contains special iteration constructions for this purpose. They can be used in one scenario only or put into the shape of iterator class that can be used in several test scenarios.

Several test scenarios can be composed in an arbitrary manner in one test script.

Implementation. The peculiarity of project using JavaTesK technology is specifications and test scenarios implementation. As ordinary software artifacts they should be implemented and debugged.

The test suite can be derived from a set of implemented specifications and scenarios automatically. Then the developer starts debugging both the tests and the implementation of the target system by running the tests.

6.4 Specification facilities

J@va uses the ADL-like specification architecture. The specifications are written in separate files that contain also auxiliary Java code, which is used for convenience and to support reuse of once written operations. Auxiliary classes and methods have no special modifiers as specification, invariant or some else. There are a variety of constructs for specifying the behavior of a single method and also the constraints on the objects' state and behavior that can involve several different methods. The specification of method in J@va can be expressed in the form of pre- and post-conditions and access constraints. The second kind of specifications is the class specifications which consist of invariants and axioms. A user is free in choice what kind of specifications to use. Each kind of them is translated into some kind of assertions and in most cases they are interchangeble, for example, all constraints can be expressed in the form of axioms and invariants can be viewed as common parts of pre- and post-conditions of all class methods.

6.5 The Specification of Methods

A method specification always is marked with **specification** modifier. It includes *access constraints*, which describes the kind of access method has to its parameters and other objects. J@va access constraints are similar to RSL ones [RAISE-language] and used to check that the method keeps the values of objects, which are read-accessible for it, and to iterate parameters while test cases generation.

```
specification class Stack {
  specification synchronized int size()
    access (read this)
  { ... }
  specification synchronized void push(Object obj)
    access (read obj; read write this)
  { ... }
}
```

Pre- and post-conditions are used to specify the behavior of the method in terms of pre- and post-states of target system. In J@va post-conditions the pre-call operator **@** and post-value operator **`** can be used. **@** has the semantics similar to ADL but it can not be applied to identifiers. All identifiers stand for pre-values of corresponding objects. To indicate the post-value of variable **x** we use **x`.** Pre- and post-conditions are used to generate oracles for specified methods. J@va has several special constructs for test coverage description. These constructs and some restrictions on post-condition structure help to make the test coverage analysis automatically.

6.6 Invariants

The classes that specify some target classes are marked with **specification** modifier and named *specification classes.* Such classes usually include *specification methods*, specifying target methods (as above considered), invariants and axioms. *Invariant* represents the constraint on the object state that must be obeyed during all object lifetime. In J@va invariants are boolean read-only methods marked with **invariant** modifier. The invariants of some class are checked for an object of this class in oracles every time the object's state is changed.

```
class Department { ... }
specification class Employee {
  public int age;
  public Employee boss;
  public Department department;

  invariant boolean I1()
  {
    return (age > 0);
  }
  invariant boolean I2()
  {
    return (boss != null) ==> (department == boss.department);
  }
}
```

6.7 Test Scenarios

The test scenarios represent a powerful facility for test development. In general, a scenario defines a model of target system called the *testing model*. Scenario must define the state class for this model and the transitions, which must be described in terms of target methods. The testing model represents a FSM, which is a factorization of the FSM representing the target system. One can find details of this approach in [Bourdonov]. In a simple case the test scenario represents the scenario of tested operations calls that can lead to some verdict on their operation. The test constructed from such scenario executes the stated scenario and checks the verdict, it also checks the results of each operation with the help of its oracle. The J@va allows to use in scenarios such constructs as iterations, non-deterministic choice and serialization. The scenario iteration mean is powerful feature. In addition to usual enumeration of iteratable values the construct builds test scenarios, which starts from the same state of testing model. For example, the code

```
iterate(int i = 0; i < 10; i++)
{
  object.method(i);
}
```

denotes that when the test finds itself in some state at first time it executes object.method(0), at second time - object.method(1), and so on until it comes in this state at eleventh time - than it has nothing to do in this state and must look for some other state of the testing model, where there exists untested operation. If it does not find such state, the test finishes.

6.8 Axioms

An axiom in J@va is represented as boolean method marked with **axiom** modifier. The axioms can be located only in a specification class. There are no restrictions on called methods and their behavior in axiom. An axiom states some general constraint on the behavior of an object of specified class. Axioms may have pre-conditions. Well known algebraic and co-algebraic specifications in general case can be reduced to the axioms, but J@va axioms provide a special, more convenient construction. We call the mixed algebraic and co-algebraic specifications simply algebraic for short. Such a specification claim that two or more chains of operations are equivalent, that is if one carries out any of this chains from one state, he will obtain the same results of last operations and the same final state. Algebraic specification in J@va can be represented as a pseudo-method in specification class marked with **equivalence** modifier and having several bodies, each realizing a chain of operations.

```
specification class Stack {
  public int size;
  public final static int MAX_STACK_SIZE
    = java.lang.Integer.MAX_INT;

  specification synchronized Object pop() { ... }
  specification synchronized void push(Object obj) { ... }

  axiom boolean A1()
  {
    pre { return (size != 0); }
    {
      int old_size = size;
      pop();
      return (size == old_size - 1);
    }
  }
  equivalence void E1()
  {
    pre { return (size != 0); }
    { push(pop()); }
    { ; }
  }
  equivalence Object E2(Object obj)
  {
    pre { return (size != MAX_STACK_SIZE); }
    { push(obj); return pop(); }
    { return obj; }
  }
}
```

Axioms serve for generation of test scenarios, which use them as test scenarios and also check their verdicts. Similar to the scenarios J@va allows to use iterations, non-deterministic choice and serialization in axioms. The **iterate** construct in axioms can be considered as typed generality quantifier.

Here are all main features of J@va, the only exception specification mechanism left out scope of the consideration. So, the J@va (from syntax point of view) might be truly considered as slight extension of Java.

6.8.1.1 Sample J@va Specifications

Below we are presenting an example of J@va specifications for bounded stack class. This example demonstrates the main features itemized in previous section.

```
import java.util.Vector;
specification class StackSpecification  {
static final public int MAX—SIZE = 2048;
final public Vector items = new Vector(MAX—SIZE); // model state
// object intergity constraint
invariant I1()
 { return items.size() ?= 0 && items.size() != MAX—SIZE; }
// auxiliary comparison method
boolean synchronized auxIsEqual(StackSpecification other)  {
if(items.size() != other.items.size()) return false;
for(int i = 0; i ! items.size(); i++)
if(items.elementAt(i) != other.items.elementAt(i))
return false;
return true;
}
// specifications of target operations
specification synchronized Object pop() access(read write this)  {
```

```
pre { return items.size() != 0; }
post {
branch "Single branch";
// x` denotes postvalue of x
// @!expr? denotes execution of the expr in prestate
return pop` == @items.lastElement() && pop` != null
&& this`.auxIsEqual(
@(this.items.removeElementAt(items.size()1)));
}
}
specification synchronized void push(Object obj)
access(read obj; read write this) {
pre { return obj != null && items.size() != MAX—SIZE; }
post {
branch "Sinlge branch";
return this`.auxIsEqual(@(this.items.addElement(obj)));
}
}
// axioms
equivalence synchronized Object A1(Object obj) {
pre { return items.size() != MAX—SIZE; }
 { push(obj); return pop(); }
 { return obj; }
}
equivalence synchronized void A2() {
pre { return items.size() != 0; }
 { push(pop()); return; }
 { return; }
}
}
```

The following test scenario example verifies the behavior of Stack in the states close to empty and full. The parameter of scenario denotes the number of calls of the target operations in such states.

```
class StackScenario implements Scenario {
final protected StackSpecification s;
// the scenario main method
scenario boolean main(int n)
 {
if(s.items.size() == 0) s.push(new Object());
else if(s.items.size() ? 1)
while(s.items.size() != 1) s.pop();
for(int i = 0; i ! n; i++)
 { s.pop(); s.push(new Object()); }
if(s.items.size() != 1) return false;
while(s.items.size() != StackSpecification.MAX—SIZE1)
s.push(new Object());
for(int j = 0; j ! n; j++)
```

```
 { s.push(new Object()); s.pop(); }
if(s.items.size() != StackSpecification.MAX—SIZE1)
return false;
 }
 }
```

6.9 Process Artifacts

Project Requirements. JavaTesK requires project requirement to be recorded in the form of use case scenarios set and J@va specifications set. This form allows easy elaboration of general requirements into as detailed ones as we need and after applying some design — transformation into complete specification suite of the target system.

Target System Model. This artifact is necessary in reengineering JavaTesK projects, especially when source code is badly commented and system documentation is out of date. Target system model can be obtained by automatic analysis of the target system by some set of reengineering tools. We cannot advise to use some specific tool here. On the contrary, our experience showed that different tools often educe different views of the target system, every of which may be important in system architecture comprehension. So, usually the target system model consists of several representations produced by different tools.

Verification Suite. The main specific for JavaTesK projects artifact is the verification suite consisting of three parts: specification suite, test scenarios, and test suite. The latter is generated from the former two with the help of J@va translator. Let us consider these artifacts in more detail.

Specification Suite. Specification suite consists of target system formal specifications written in J@va specification language. The most part of it is a set of J@va specification classes representing the full functionality of units in the target system.

J@va specification class represents the precise description of a chunk of functionality implemented in the target system. This functionality can be implemented by one object or by a group of linked objects.

State is specified by means of class attributes and state constraints, represented as invariants. Behavior is specified by the set of methods' specifications, which determine the sufficient conditions that must hold for given method to work properly (pre-condition) and what is its proper behavior (post-condition, expressed in the form of constraints on the system state before method call, the parameters of method call and the system state after the method ends).

Specification classes can use other specification classes or Java library classes. The important (but technical) restriction is that specification class can extend only another specification class and cannot extend Java class. The extension between specification classes can be specified as functional or LSP-compliant (Liskov's Substitution Principle).

The only new kind of relationships (new in Java-world) between specifications is *refinement*. The refinement can be used to improve reuse of the specifications. It allows to create a specification library for some problem domain and to use it while specifying any application in this domain. Such development strategy can dramatically decrease the labor-intensiveness of specification development.

Another part of specification suite is mediators. Mediator is a special J@va class, which main purpose is to link some specification class with corresponding implementation. Mediator should translate the calls of specification methods into the corresponding calls of target methods and to build the post-state of specification class on the base of the state of implementation.

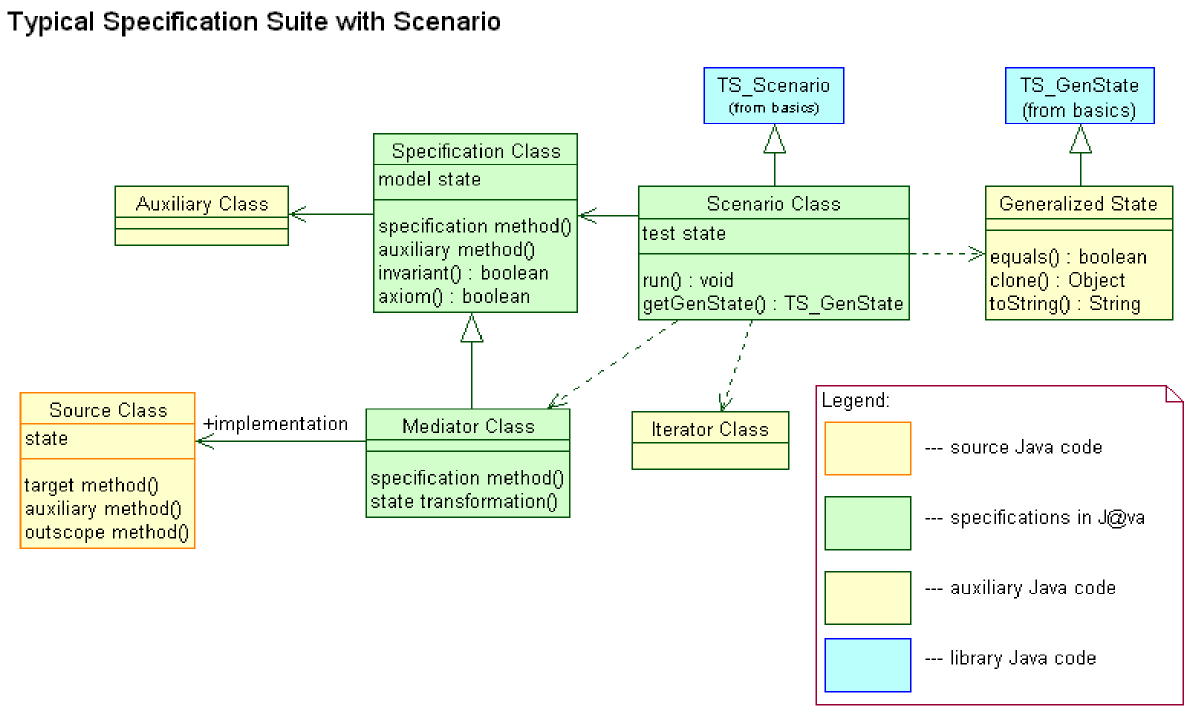

Figure 1.

Test Scenarios. Test scenario is a special J@va class describing a sheaf of possible calls of some target methods. We use the word “sheaf” here because the control flow of scenarios can be arbitrary complex, every path in it should determine some sequence of target methods.

Test scenario may use its own model of the target system state called the *testing model*. For ease of scenario development and its correspondence with automated testing there exist predefined Java interface of testing model state called **`TS_GenState`**. The testing model must be recorded as Java class implementing this interface. Test scenario itself must implement predefined **`TS_Scenario`** interface.

Test Suite. The test suite consists of a part generated from specifications and scenarios by J@va translation to Java and some predefined part, which is represented by a steady part of test system and supporting classes for generated ones.

The generated part structure is as follows.

Every specification class gives rise to *model class*, *oracle class* and *coverage information classes* (for each specification method one coverage information class is generated).

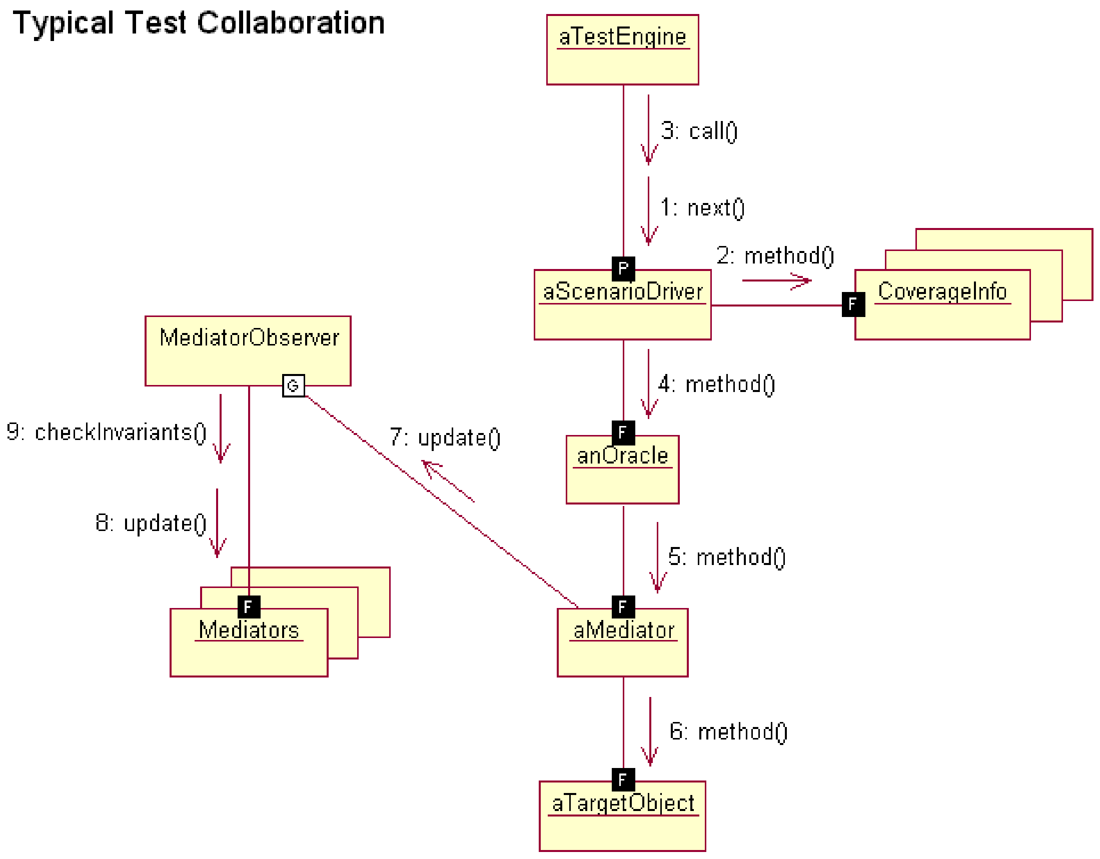

Figure 2.

Model class is Java class having the interface and the attributes specified in corresponding specification classes. It also has special methods for checking all the specified invariants. Oracle class is Java class having methods corresponding to specification methods of the specification class. Oracle object is linked with some model object, which behavior it checks. Each oracle method must check pre-condition of corresponding specification, then call the model method and then check the post-condition.

Coverage information class object is intended to store information on coverage obtained according to the chosen criteria. This information is used for run-time optimization of test — if the next call of some target method cannot obtain better coverage, it can be skipped.

Each J@va mediator class for some specification produces corresponding Java mediator class, which extends the model class for the same specification. Object of this class is used as model, which behavior is checked by the oracle object. Mediator defines specification methods in terms of implementation, so the oracle object checks implementation behavior in the long run. Another responsibility of the mediator object is to make its state consistent with implementation after every call.

Scenario class gives rise to *scenario driver class* in Java. This class has several standard methods helping it to cooperate with predefined so-called *test engine class*. The collaboration of test engine object and scenario driver object constitute the test sequence generation mechanism. This mechanism is based on state-transition graph traversal algorithm, where test engine implements the common part of algorithm and scenario driver the variable one concerning the description of possible states and transitions. There are several used algorithms designed for different graph types, each implemented in different test engine class.

Figure 2. shows the sequence of calls between test suite components corresponding to one target method call in typical test (the calls are enumerated as 1, 2,…). At first test engine object **`aTestEngine`** calls **`next()`** method in scenario driver object **`aScenarioDriver`** to determine which next call will enlarge the coverage obtained until this moment. Scenario driver uses corresponding **`CoverageInfo`** object to get information about already obtained coverage, iterates target method calls until it finds the necessary one, then it stores the information about this call and returns true to the engine. If scenario driver does not find a suitable call, it returns false and the engine uses underlying traversal algorithm to find a path to another state.

When the suitable call is found the engine calls **`call()`** method in scenario driver and this call is translated through oracle and mediator to actual call to implementation. After the end of implementation method mediator object invokes the static method **`update()`** in **`MediatorObserver`** class. This method in turn invokes updates in all mediator objects — this makes their states consistent with target system state — and then checks invariants in all mediator objects to make sure that the call of target method does not break any of them.

After that the control is returned to the oracle, it checks the post-condition of the method called and return the control to the engine through scenario driver.

Such activity goes on until the necessary coverage obtained or neither of target method calls in either state can enlarge the coverage.

• Conclusion

Feasibility in industrial context. Traditional niche for formal specification is high-level design in forward engineering processes. Now application of such approach in industrial context is restricted because software engineers need more detailed specifications. Reverse engineering of legacy requests especially strong requirements to accuracy of the specifications. These additional requirements to specifications and common problems of test generation explain position of skeptics said impossibility of deploying specification based testing in industry.

RedVerst experience has shown feasibility of the approach. During 6 last years RedVerst has produced over 200 Kline formal specification in RSL and has developed tools for specification support and test generation. Size of generated test suites is over 10Mline in target programming languages. RedVerst has introduced 2 kinds of software verification processes. First one is intended for legacy reverse engineering and improving, second one is regression testing including maintenance of formal specifications and test suites maintenance. RedVerst has demonstrated high degree of specifications and test design artifacts re-use. The high re-usability causes advantages of specification based testing for a long term prospects, and, in particularly, in case of regression testing.

Reason of specification based testing lag. In industry specification based testing falls behind traditional "white box" testing. There are a few reasons causing the situation. First ones are as follows:

- the lack of personnel with skills in formal specification languages;
- the lack of test design skills;
- the lack of tools;
- the limitations on the software development process structure and timing.

There are a few other lacking issues causing the troubles of specification based testing deployment. The detail analysis of current situation in specification based testing is provided in J.Ryser's technical report [Ryser].

How to introduce specification based testing innovations. The following are necessary for successful deployment of formal methods in industry development processes:

- A presence of a **critical mass** of capabilities and features of notations, techniques, and tools.
- Industry needs both specific formal **specification languages** and **specification and test design extensions** of programming languages. First ones facilitate study and first phases of deployment of formal techniques, second ones will used as notation massive use.
- A **close integration** with widely used Software Development Environments (SWDE). The instruments for the specification and testing must be tightly integrated with (or, better, be an integral part of) any SWDE.

Considering the conditions needed to achieve the critical mass, we should note that the lack of some of the listed features should not deter the deployment of formal methods. Rather a gradual, incremental accumulation of the features is viewed as a preferred strategy.

There is no real sense to directly overcome resistance of software engineers. We should do not make to use specification based approach, indeed we should provide tools and techniques that facilitate their work, that help them to do good work.

At the same time there is a favorable obstacle that facilitates deploying the specification based testing into practices. Testing is usually considered as tedious, fatigue, wearisome work. It is true, but why do we not apply this appreciation to software design and development - the work includes a lot of routine activities too. The root of the problem is creative nature of a design as a whole. The specification based testing requires first to model a system under test. A test designer can (and must) compete with software designer. They stand on parity positions, both test and software designers present their own models of a target system and testing only demonstrates coincidence or differences of their visions. This human factor will play important role in propagation of the specification based testing to practice.

Acknowledgements

We are grateful to RedVerst team and Alexander Vorobiev (Nortel Networks), he reviewed early version of the paper and participated in discussion on deployment of advanced techniques in industry processes.

7 References

[ADL] http://www.sun.com/960201/cover/language.html

[Antoy&Hamlet] S.Antoy, D.Hamlet. Automatically Checking an Implementation against Its Formal Specification. IEEE trans. On Soft.Eng, No.1, Vol.26, Jan. 2000, pp.55-69.

[Bjorner] D.Bjorner. Formal specification is an experimental science. Programmirovanie, No. 2, Moscow. English version in Programming and Computer Software, ISSN 0361-7688, N-Y, Vol. 20, No. 2, March-April, 1994.
[Bourdonov] I. B. Burdonov, A. S. Kossatchev, V. V. Kuliamin. Application of Finite Automatons for Program Testing.- Programming and Computer Software, Vol. 26, No. 2, 2000, pp. 61–73.
[Clarke&Wing]E.M. Clarke, J.M.Wing. Formal Methods: State of the Art and Future.- http://www.cs.cmu.edu/afs/cs/usr/wing/www/mit/paper/paper.ps
[Doong&Frankl] R.-K.Doong, P.Frankl. Case Studies on testing Object-Oriented Programs, Proc. Symp. Testing, Analysis, and Verification (TAV4), 1991, pp.165-177.
[Eiffel] http://www.eiffel.com
[ESI-black box] ESSI Std 610.12-1990.
[ESI-verification] http://www.esi.es/Help/Dictionary/Definitions/Verification.html
[FME] http://www.fme-nl.org/fmadb088.html
[iContract] R.Kramer. iContract – The Java Design by Contract Tool. Fourth conference on OO technology and systems (COOTS), 1998.
[ISPRAS] http://www.ispras.ru/
[Jones] C.B. Jones. Systematic Software Development Using VDM. Prentice Hall International, 1986.
[KVEST] I.Bourdonov, A.Kossatchev, A.Petrenko, and D.Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. Proceedings of World Congress of Formal Methods, Toulouse, France, LNCS, No. 1708, pp. 608-621, 1999.

[KVEST2000] A.Petrenko, A.Vorobiev. Industrial Experience in Using Formal Methods for Software Development in Nortel Networks.- Proc. Of the TCS2000 Conf., Washington.,DC, June, 2000.

[Larch] J.Guttag et al. The Larch Family of Specification Languages. IEEE Software, Vol. 2, No.5, September 1985, pp.24-36.
[Murray] 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
[OMG] http://www.omg.org
[Peters&Parnas] D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering*, Vol. 24, No. 3, pp. 161-173.
[RAISE-language] The RAISE Language Group. The RAISE Specification Language. Prentice Hall Europe, 1992.
[RUP] http://www.rational.com
[RedVerst] http://www.ispras.ru/~RedVerst/
[Ryser] ftp://ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.03.pdf
[VDM_SL] N.Plat, P.G.Larsen. An Overview of the ISO/VDM-SL Standard. SIGPLAN Notes, Vol. 27, No. 8, August 1992.
[VDM++] http://www.csr.ncl.ac.uk/vdm/

Standards

[MSC] Message Sequence Charts. ITU recommendation Z.120.

[SDL]	Specification and Design Language. ITU recommendation Z.100.
[TTCN]	ISO/IEC 9646-3. Tree and Tabular Combined Notation (TTCN).