Testing automation of projects in telecommunication domain

Alexey Veselov, Vsevolod Kotlyarov
Saint-Petersburg State Polytechnic University, Saint-Petersburg, Russia
a.veselov@ics2.ecd.spbstu.ru, vpk@ics2.ecd.spbstu.ru

***Abstract* – This paper presents an integrated approach to testing automation of telecommunication projects along with proposals to automation of conformance testing. The underlying idea is to benefit from combining formal verification and testing automation techniques in order to improve product quality.**

I. INTRODUCTION

Testing is known to be an essential step in any modern industrial software development process. The importance of software testing and its impact on software quality can't be underestimated. To date, spending up to 40% of efforts on testing is not uncommon for a software company. Therefore, testing automation and testing efficiency – the issues addressed in this paper - are very important.

This paper describes a part of technology which combines formal verification, testing and code generation techniques in order to improve software quality, reduce efforts and costs. This technology covers the whole software development process which starts from the initial requirements and goes through formalization, verification, simulation, code generation and testing to a software product. The technology's part, which is described below, relates to the techniques which on the one hand can be used for automation of functional tests generation, and on the other hand can be applied for conformance testing.

The underlying idea of the proposal is to benefit from presence of verification step in the technology. Verification is a powerful technique which can significantly improve software quality and formally prove absence of mistakes in algorithm. But it also has another important advantage: a model is created on the verification step, and each path in the model's behavior tree can be considered as a potential test case. The number of such paths in industrial-scale projects is enormous, so a special filtering technique is required to obtain suitable (optimal) test suite. This approach is applicable not only to automation of functional tests generation, but also for automation of conformance testing (in case if a model of the corresponding standard is provided). An example of such standard is CDMA2000 [1]. CDMA2000 is a family of 3G mobile technology standards which use CDMA (Code Division Multiple Access) channel access to send data, voice, and signaling data between mobile phones and cell sites. Importance of compliance of any CDMA2000 device and cell site with this standard is obvious. In practice, compliance is checked by rather significant amount of conformance tests which are prepared manually. Proposed approach to conformance testing allows avoiding manual tests development by means of reusing artifacts of verification step.

This paper outlines the main principles of automated test suite generation on the base of formalized specifications in the language of basic protocols using the verification tool VRS [2, 3] and tests generation tool TAT [4].

II. TECHNOLOGY OVERVIEW

Despite the paper is focused on testing automation part of the technology, overview of the whole technology chain is required for proper understanding of application domain of the proposing ideas.

The technology is intended for automation of manual efforts in software development process as much as possible (by applying code generation techniques) and increasing products' quality (because of combining formal verification and testing).

According to the technology chain, illustrated in Fig. 2, software development starts from understanding and formalizing of the requirements. On the formalization step, engineers create a model of the system in terms of basic protocols. Basic protocol is a formal description of some action which shall be performed if the system goes into a certain state (pre-condition is satisfied), and a new state of the system after performing this action (post-condition). In other words, basic protocol is a "small piece" of system's behavior. Basic protocols are represented in the following notation:

$$\forall X : \alpha(X) \xrightarrow{\mu(X)} \beta(X)$$

where X is protocol parameters' list, α is a pre-condition, β is post-condition and μ is action; α, μ and β may depend on X. This is a Hoare's triplet [5].

The formalism of basic protocols is based on the theory of agents and environments with the insertion function [6]. It was proposed by A. Letichevsky et al. for creating system behavior models suitable for automated verification [7, 8]. Basic protocol is just a small MSC chart. Fig. 1 illustrates a basic protocol example.

Two basic protocols can be concatenated when post-condition of the first one is equal to pre-condition of the second one. All possible concatenations of basic protocols construct the model's behavior tree. Each path in this tree is a possible scenario of system's behavior.

Formalization is mostly manual work, but it can also be automated or simplified in some sense. For example, the user can perform formalization in terms of use cases (Use Case Maps (UCM [9]) or Message Sequence Charts (MSC [10])) or an UML [11] model. Special tools are used for basic protocols generation from UCM, MSC and UML.

The second step of the technology chain is verification. The system's model, represented in terms of basic protocols, is

verified against some properties using model checker of the VRS tool.

A set of basic protocols can be interpreted as a set of states and transitions, i.e. as a state machine. Having a state machine which describes system's behavior, it is possible to generate system's logic code in a target language. But there is no tool which can generate executable code from basic protocols, so intermediate representation is required. Specification Description Language (SDL [12]) was chosen to be the language of this intermediate representation. SDL is rather simple and well-known language for describing state machines. Presence of intermediate language makes it much more convenient for the engineers to debug models.

So, the third step is conversion from basic protocols representation to SDL representation. This conversion is performed by special tools.

As soon as SDL code is obtained, it is possible to generate code in target language. Generation of target code is the fourth step. A user has opportunity to choose any SDL to target code translator, but in our practice we use Sdl2cpp because it is seamlessly integrated into the technology chain. Sdl2cpp tool is efficient and reliable translator from SDL to C++. Sdl2cpp can generate configuration file for TAT (Test Automation Toolset) with test environment description. So, in case of using Sdl2cpp together with TAT, test environment configuring becomes transparent for a user (no manual efforts required).

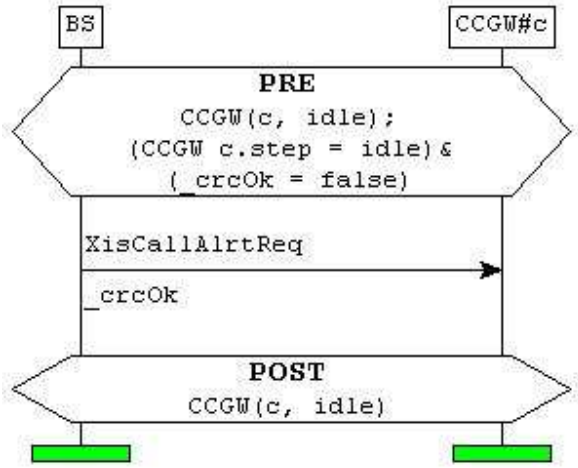

Fig. 1. Basic protocol

As soon as model's executable code is obtained and test environment is configured, it is possible to run tests and perform real-time simulation. It is also rather convenient to do step-by-step simulation in IBM Rational SDL Suite [13] with tracking of system's behavior in graphical mode.

The fifth step is creation of test coverage criterion. Model in terms of basic protocols is a behavior tree with enormous number (more precisely – infinite number) of potential scenarios. It is impossible to cover this entire tree with tests, so a filtering criterion is required. The user is free to choose any criterion, but usage of requirements coverage criterion makes it is possible to generate relatively small test suite which covers functional requirements. This criterion requires creation of "chains" for each requirement. "Chain" is a sequence of key impacts on the system under test (SUT) and systems' responses which cover a particular requirement (requirement may be covered by more than one "chain").

Creation of chains is a process of requirements interpretation, so it is mostly manual work, especially if requirements are presented in natural language. Chains creation is significantly simplified if customer provides requirements in formal notation, for example in terms of UCM or MSC.

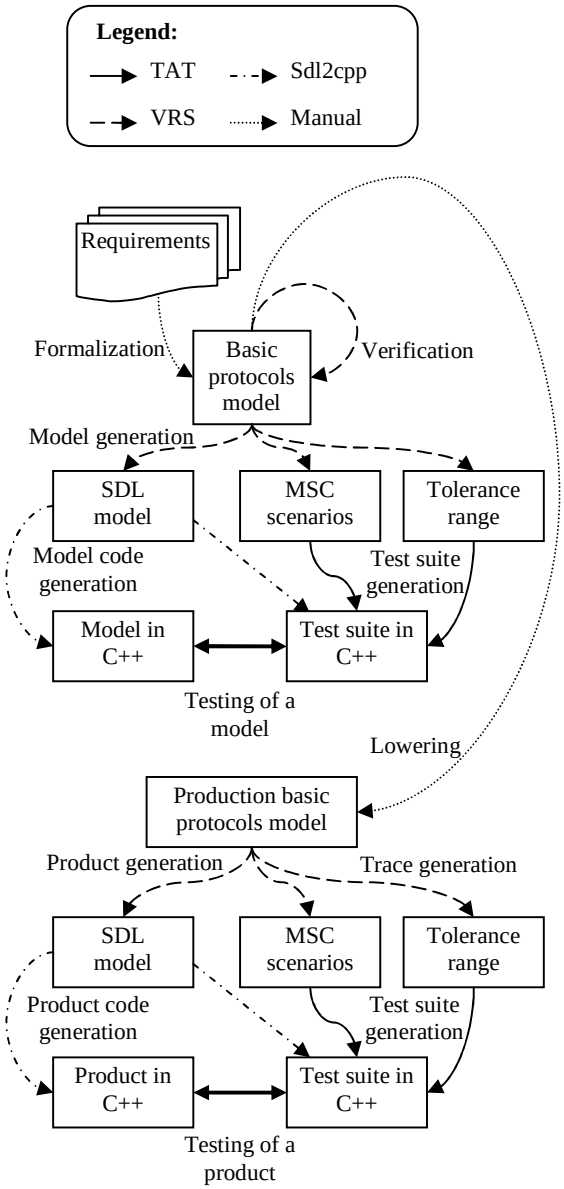

Fig. 2. Overall technology scheme

The sixth step is test scenarios generation. VRS tool uses chains for guided search in model's behavior tree and generates a trace if it can find a path with chain's key events. As a result, a set of scenarios which cover functional requirements is generated. These scenarios are represented in terms of MSC charts. For the requirements which for some reasons can't be expressed in terms of chains (for example,

non-functional requirements), test cases shall be created manually.

TAT tools are used for generation of executable test environment code from MSC charts. So, the next (seventh) step is generation of a test suite in a target language using TAT tools. This step also includes substitution of symbolic parameters of MSC scenarios with actual values taken from tolerance range.

The eight's step is running generated test suite against the model.

The technology chain consisting of the mentioned eight steps works fine when scale of a project is rather small. But when the project starts to grow, verification becomes a bottle neck because of the state explosion problem. Verification of industrial-scale projects requires abstracting from some details, therefore basic protocols model must be detailed after verification and before production code generation. It is usually required to maintain two models within the scope of one project: high-level model which is used for verification, modeling and simulation, and production model which is used for product code generation. Each model requires its own test suite.

So, the ninth step is detailing. This step is also known as lowering because the model is filled with low-level details. Usually it implies adding signals' parameters, replacing functional stubs (when a complex behavior is encapsulated in one signal; e.g.: usage of "InitSession" signal instead of describing real sequence of messages for session initialization) with actual sequence of signals and adding omitted signals. Lowering is another one manual part of the technology.

As soon as the model is filled with low-level details, it can be used for product code generation. To generate product code, the user has to repeat steps 3 and 4 for the detailed model. To generate a test suite for the product, the user has to repeat steps 6 and 7 (step 5 is not required because chains for the high-level model are applicable to the detailed model).

III. TESTING AUTOMATION

The whole technology chain consists of several steps. Steps 5, 6, 7 and 8 relate to tests preparation, generation and execution activities.

Tests preparation starts from requirements interpretation and formulating coverage criteria in terms of "chains" – sequences of key events. These chains are used as oracle for guided search in model's behavior tree. It is natural that from infinite number of possible system's behaviors, test engineers want to choose reasonable quantity of scenarios which cover SUT requirements. This result can be achieved in case of using chains: VRS tool will automatically generate corresponding traces (test cases). In the worst case, the number of test cases will be equal to the number of chains, but in practice some traces cover more than one requirement, so quantity of tests is usually less than quantity of chains. Special tool is responsible for selecting minimal test suite from generated traces. This tool also marks the points (in MSC traces), where coverage of each particular requirement starts and ends, along with the places of key events.

As soon as a set of scenarios is selected according to the coverage criteria, it is required to replace scenarios' parameters with concrete values (if they contain symbolic parameters).

VRS tool can work in two modes. The first one is regular model checking where all the parameters have concrete values. The second mode is called "symbolic"; in this mode tolerance range for each parameter is evaluated after applying each basic protocol, so generated MSC charts (scenarios) contain symbolic parameters, and VRS provides their values' constraints (tolerance ranges). So, in case of using "symbolic" mode, there is one additional step: replacement of MSCs' symbolic parameters with concrete values taken from tolerance range.

One MSC with symbolic parameters is actually a set of equivalent behaviors (the same sequence of events for any parameters' values from tolerance range). So, taking into account parameters' constraints and dependencies between parameters, the user can obtain a set of test scenarios with same sequence of events, but different parameters' values.

A set of concrete values for each trace we will call a profile. The user is free to choose any profile, but usually the following profiles are chosen:

- "left edge" profile - all independent parameters have minimum possible values (from the tolerance range);
- "right edge" profile - all independent parameters have maximum possible values (from the tolerance range);
- "middle" profile – all independent parameters have arbitrary values (from the tolerance range);
- "error" profile – at least one parameter has value out of tolerance range. Test case with any "error" profile shall always fail.

Scenarios with concrete parameters can be used for test suite generation.

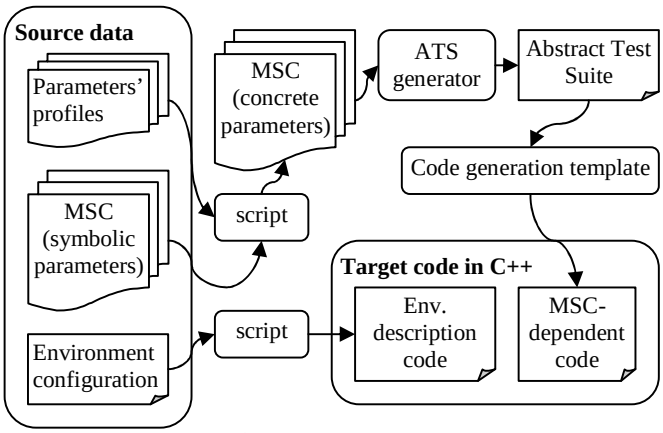

Fig. 3. Test suite generation

Test Automation Toolset (TAT) is used for generation test suites in target language on the base of scenarios represented in terms of MSC charts and configuration XML files with environment description.

Test suite generation consists of several major steps:

- Analysis of XML configuration files and generation of environment-specific code in target language;

- Generation of Abstract Test Suite (ATS) – representation of a test suite as a set of states and transitions between them (state machine) in TCL. ATS is generated on the base of MSC charts;
- Generation of a test suite in target language on the base of ATS.

These steps are illustrated in Fig. 3

TAT has very flexible mechanism of code generation templates. Using templates, TAT can be configured for generation of test suite code in any language. Currently, the most powerful template is for C++, but several other languages are also supported, including Java SE, Java ME and TTCN-3 (only C++ code generation is considered in the scope of this paper).

Generated test suite interacts with a system under test via IP sockets (both TCP and UDP are supported) or via Unix domain sockets (both stream and datagram are supported). A user can specify which sockets to use in TAT configuration files along with the following settings:

- Initialization code which will be executed before the beginning of each test case. Sometimes some initialization of a system under test is required before running a test case. The code which is responsible for moving SUT to its initial state shall be placed in corresponding section of TAT XML configuration file (initialization section).
- Finalization code which will be executed after a test case is finished regardless of the verdict (pass/fail). It is a good practice to free resources when they are not in use any more, this can be done in corresponding section of TAT XML configuration file (finalization section).
- Timeouts of waiting for incoming signals. The user can specify how much time to wait for signal(s) in milliseconds. If test environment does not receive the signal(s) which it expects (according to the MSC scenario) within this time interval, corresponding test will be marked as failed. It is possible to specify default timeout for all incoming signals and, if necessary, redefine this parameter for any particular incoming signal.
- Signals' distinguisher code (if not provided by Sdl2cpp). This code is responsible for distinguishing incoming messages. In case of using TAT together with Sdl2cpp, this code is generated automatically. If TAT is used as stand-alone tool, the user has to provide this code manually.
- Signals' sending/receiving code (if not provided by Sdl2cpp). The user may provide own code for serialization and deserialization of messages and their parameters. In case of using TAT together with Sdl2cpp, this code is generated automatically. If TAT is used as stand-alone tool, the user has to provide this code manually.
- Instances and interfaces definition (if not provided by Sdl2cpp). All the instances (used in MSCs) must be described in TAT XML configuration file along with their interfaces (port numbers for IP sockets or file names for Unix domain sockets). Instances may be of two types: "env" (abstraction of test environment) and "model" (abstraction of system under test). In case of using TAT together with Sdl2cpp, this description is generated automatically. If TAT is used as stand-alone tool, the user has to provide it manually.
- Message delimiter (if not provided by Sdl2cpp).
- User-defined log format. TAT has two built-in formats of logs: logs in terms of MSC charts and plain text logs. But, if required, the user can define own log format.

TAT has seamless integration with Sdl2cpp: Sdl2cpp can not only generate target code from SDL, but also completely configure TAT for testing the generated code. But even in case of using TAT together with Sdl2cpp, manual adjusting of test environment remains possible because TAT configuration can be split into two parts (stored in separate files): generated by Sdl2cpp and manually provided. Manually provided part has higher priority, so default environment settings (generated by Sdl2cpp) can be redefined by the user (if required).

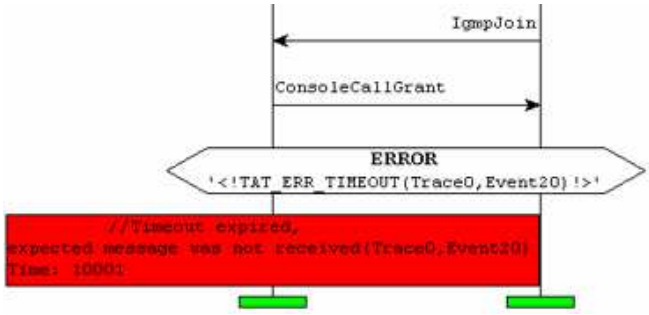

Fig. 4. Fragment of MSC log file with an error indication

As soon as a test suite in target language is generated by TAT, it can be used for testing SUT. The results of testing can be observed in log files (an example of a log file fragment in MSC format is represented in Fig. 4).

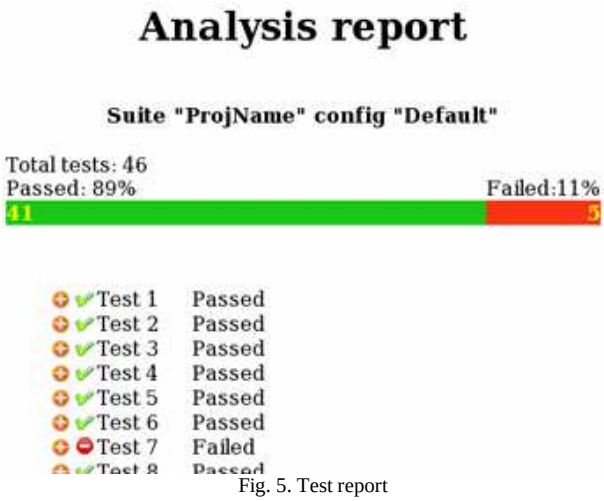

Fig. 5. Test report

TAT tool called Offline Test Results Analyzer (OTRA) is responsible for "offline" test results analysis. It compares source MSC charts (the charts which were used for test suite

generation) with logs in MSC format and generates a test report in HTML format. Fig.5 illustrates a test report example.

Each line in a test report is a link to another HTML page with comparison details and failure description (if corresponding test failed).

Described approach of automated test suites generation on the base of formal specifications can considerably reduce efforts and time of test cases preparation. Of course, this approach is not applicable to covering any requirement because sometimes requirements can not be expressed in terms of MSC charts or a sequence of key events. So, test engineer must understand the scope of its applicability and create tests for the not covered requirements manually. But, in practice (especially in telecommunication domain), majority of the requirements (especially functional requirements) can be automatically covered by tests in case of applying this technology.

Automation of functional tests generation is a powerful feature of the described testing automation approach, but it can also be applied for checking that interaction between SUT and environment takes place in compliance with some standard. It is well-known that in telecommunication domain compliance with a standard is at least not less important than checking functional requirements.

The underlying idea of the proposals about automation of conformance testing is similar to the idea of automation of functional testing, but has some peculiarities.

IV. AUTOMATION OF CONFORMANCE TESTING

Conformance testing is concerned with the assessment of the extent to which an implementation of system conforms to a specification. [14]

The process of interoperability testing and conformance testing is usually time consuming and expensive. However it is essential because the cost of releasing a product (especially in telecommunication domain) that does not operate correctly is very much higher.

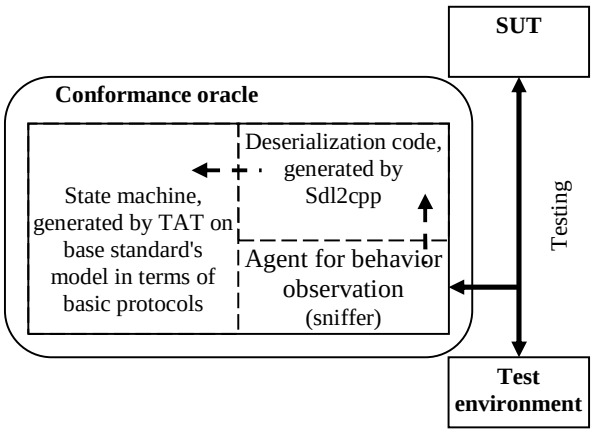

Fig. 6. Conformance oracle

And the problem here can be formulated as follows: if I have a SUT which interacts with test environment or with other system(s), how can I check whether this interaction takes place in accordance with some standard or not?

It is not easy, but very important question.

Another important question is: “Are there any inconsistencies in the standard’s specification?”

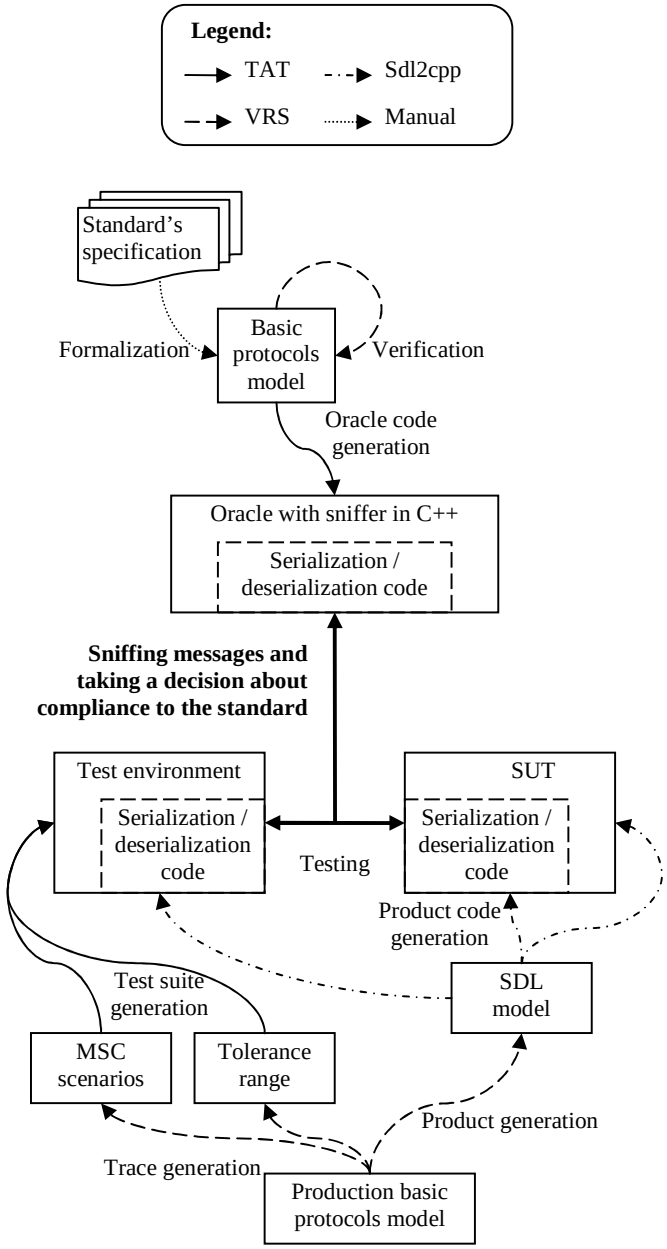

Fig. 7. Conformance checking approach

The proposing approach, illustrated in Fig. 7, tries to answer both of them. The first its step, verification, can answer the second question. Verification of a standard is not an easy task. Formalization of the standard’s specification is the most effort-consuming part of this step, and it can take much time. Unfortunately, formalization is a process which in common case can hardly ever been automated (because it is a process of requirements understanding and interpretation). However, verification can not only formally prove absence of errors, but also help us answer the first question.

In case of using notation of basic protocols, formalized model is represented as a set of states and transitions between them, i.e. as a state machine.

This state machine can be used as an oracle if place it between SUT and test environment or between SUT and other system(s) (which conform the standard).

So, the second step is generation of target code of this state machine and using it together with an agent for behavior observation (for example, a network sniffer for telecommunication domain). TAT code generation engine is going to be used for generation of the state machine code.

Normally, when Sdl2cpp generates SUT's target code from SDL, it also generates serialization and deserialization functions for each message which can be sent to the environment or received from the environment. These functions are both used on the SUT side and on the test environment side for encoding and decoding messages. So, it is not required for the user to define messages' structure (and corresponding serialization/deserialization functions) twice. In other words, Sdl2cpp can automatically configure TAT and share encoding/decoding code with it. This makes message exchange transparent for the user who works on the "signals" level of abstraction and is not concerned about what is actually sent via sockets. However, observation of socket-level messages remains possible. The same idea is also applicable to the oracle which, of course, requires a means of mapping sockets-level messages to high-level signals.

So, conformance oracle will consist of the following parts (as shown in Fig. 6):

- State machine generated by TAT on base of standard's model in terms of basic protocols;
- Deserialization code generated by Sdl2cpp;
- Agent for behavior observation.

The third step is testing. On the testing phase SUT and test environment communicate as usual (through signals exchange via network). And the only difference is presence of the conformance oracle which intercepts all the signals using network sniffer, decodes them using deserialization functions (provided by Sdl2cpp) and takes a decision about compliance with the standard.

Decision about compliance with the standard is taken in accordance with the following algorithm:

1. The oracle intercepts a message and tries to decode it. If the message can't be decoded, it does not conform to the standard.
2. If the message is successfully decoded, the oracle checks existence of any transition from its current state where condition of the transition is receiving this signal. If there is no such transition, the system does not conform to the standard.
3. Then the oracle checks signal's parameters. If parameters' values are within admissible range, the oracle performs transition to the next state and waits for the next signal; otherwise, the system does not conform to the standard.

The more test are executed with the conformance oracle, the more the user is confident that SUT conforms to the standard.

V. CONCLUSION

The developed approach of testing automation was successfully applied in several large-scale projects in telecommunication domain.

The maximum benefit (cost and time savings; increase of quality) from this approach can be achieved in case of using it as a seamless part of the overall technology, represented in section 2. But usage of TAT as a stand-alone test suite target code generator from formal specifications remains possible, so described testing automation proposals are also applicable to other software development processes.

Conformance testing automation proposals seem to be very promising because in spite of high complexity of standards formalization, automation of conformance testing together with standards verification can significantly increase product quality.

REFERENCES

[1] CDMA Development Group - http://www.cdg.org/

[2] Drobintsev P.D. Integrirovannaia tehnologia obespechenia kachestva programmnih produktov s pomoshiu verifikacii i testirovania. Kand. dis., SPbGPU. 2006. 238 p.

[3] Letichevsky A., Kapitonova J., Letichevsky Jr., A., Volkov V., Baranov S., Weigert T. Basic protocols, message sequence charts, and the verification of requirements specifications, Computer Networks: The International Journal of Computer and Telecommunications Networking, v.49 n.5, p.661-675, 5 December 2005.

[4] TAT User's Manual © 2001-2005 MOTOROLA.

[5] Hoare C.A.R. Communicating sequential processes, Prentice Hall, London, 1985.

[6] Letichevsky A.A., Kapitonova J.V., Volkov V.A., Vyshemirskii V.V., Letichevsky Jr. A.A. Insertion Programming // Cybernetics and Systems Analysis, Volume 39, Issue 1 (January 2003), p.16-26.

[7] Letichevsky A.A., Kapitonova J.V., Volkov V.A., Letichevsky Jr A.A., Baranov S.N., Kotlyarov V.P., Weigert T. System Specification with Basic Protocols // Cybernetics and Systems Analysis, Volume 41, Issue 4 (July 2005), p.479-493.

[8] Baranov S., Jervis C., Kotlyarov V., Letichevsky A., and Weigert T. Leveraging UML to deliver correct telecom applications in UML for Real: Design of Embedded Real-Time Systems by L.Lavagno, G. Martin, and B. Selic (editors), pp. 323–342, Kluwer Academic Publishers, 2003.

[9] Recommendation ITU-T Z.151. User requirements notation (URN), 11/2008.

[10] ITU Recommendation Z.120. Message Sequence Charts (MSC), 11/99.

[11] OMG Unified Modeling Language - http://www.omg.org/spec/UML/2.2/

[12] ITU-T Recommendation Z.100, CCITT Specification and Description Language (SDL), 03/93.

[13] IBM Rational SDL Suite - http://www-01.ibm.com/software/awdtools/sdlsuite/

[14] ETSI Conformance Test Specification - http://portal.etsi.org/mbs/testing/conformance/conformance.htm