Validation and Testing

DIETER HOGREFE, BEAT KOCH AND HELMUT NEUKIRCHEN

Dieter Hogrefe (42) graduated with a diploma degree in Computer Science and Mathematics from the University of Hannover where he later received his PhD. His research activities are directed towards Computer Networks and Software Engineering. He has published numerous papers and two books on analysis, simulation and testing of formally specified communication systems. He has worked for SIEMENS research centre, the University of Hamburg, and the University of Berne. Since 1996 he has been director of the Telematics Institute in Lübeck and full professor at the University of Lübeck.

hogrefe@itm.mu-luebeck.de

Beat Koch (35) studied Computer Science at the University of Berne, Switzerland. After his graduation in 1994, he worked as a software engineer in industry. Since 1996, he has been a research assistant at the Intitute for Telematics of the Medical University in Lübeck, Germany. His research focuses on automatic test generation and he is one of the developers of Telelogic's Autolink tool. Currently, he is finishing his PhD thesis on "Test-purpose-based Test Generation for Distributed Test Architectures".

bkoch@itm.mu-luebeck.de

Use of formal specifications provides the basis for allowing validation of the specification towards expected behaviour, and it allows testing of an implementation according to the formal specification. This paper introduces and provides an overview of techniques for validation and testing.

Introduction

When formal description techniques (FDTs) such as the *Specification and Description Language SDL* [17] were developed for computer communication systems, the idea of precise and rigorous specification were the driving force. Prose specifications of complex technical matters are not always sufficient to guarantee interworking of products from different vendors. Also, hidden ambiguities are sometimes only discovered after the respective systems have been put into operation. Errors of this kind tend to be very expensive. FDTs have promised some help in this matter, because they make possible rigor, precision, completeness, and so on.

However, it did not take long before these promises proved to be only partly true. While FDTs provide the basis for rigor, precision and completeness, they do not guarantee these qualities. In fact, using FDTs is very similar to programming, from which it is known that errors occur easily. Hence, *validation* techniques are very important and will be explained in the second section of this paper.

Even if we have a perfectly valid specification, its value is somehow limited if we cannot relate it formally to an implementation. In some cases, this relationship comes for free: If the specification is directly translated into an executable implementation without any manual steps. Unfortunately, this is not the common case. Usually, the SDL specification abstracts from a number of details which later appear in an implementation. These abstractions are filled with life during the design and implementation process. That is of course a source for errors. We will therefore devote the third section of this paper to the aspect of conformance testing, i.e. to the question on how to show that an implementation really conforms to a specification.

We conclude this paper with a summary and a view on current research activities.

Validation

Specifying a system with a formal specification language like SDL [16] is in some respects similar to implementing a system with an ordinary programming language. In both cases, the result will probably contain errors. Therefore, formal specifications as well as system implementations have to be tested in some way. However, although it is basically the same problem, the testing of an implementation is different from the testing of a formal specification.

On a terminology level, implementation testing is also called *verification* and it involves activities concerning the question "Am I building the system right?". In contrast, *validation* deals with the question "Did I build the right system?" [21].

In the domain of designing and testing computer protocols using formal methods, both terms are used in a mixed way. This is due to the fact that a strict distinction of these two testing terms is not always possible.

Hence, according to [12], the term validation is used to describe all activities "used to check that the formal specification itself is logically consistent".

Another difference between the testing of an implementation and a specification is the underlying machine. The model specified by the formal language is usually executed on an abstract machine with potentially infinite resources. Furthermore, several views of one system may be specified using different models or even different specification languages. This is in contrast to the testing of an implementation module, which comprises all the different models and runs on a physical machine.

There are several methods to check the consistency of a formal specification. Static techniques analyse the formal specification without executing the model described by the specification. Dynamic analysis techniques build an executable model from the formal specification in order to validate it.

Helmut Neukirchen (29) studied Computer Science at the University of Technology Aachen with a focus on distributed systems and software construction. He graduated with a diploma degree in 1999. Since January 2000 Helmut Neukirchen has been research assistant at the Institute for Telematics at the Medical University of Lübeck on the EU project INTERVAL. His main research interests are formal methods in specification and testing of communication and real time systems.

neukirchen@itm.mu-luebeck.de

Static analysis

Static analysis techniques evaluate a specified model without executing it. This can be achieved manually by reviews, inspections or walk-throughs, and in an automated way by syntax and semantic analysis.

Depending on the kind of manual static analysis, the author, other experts or professional inspectors analyse the documents by following a checklist. Besides general questions regarding completeness and consistency, such checklists may contain questions concerning properties specific to the formal language used for specification.

In the domain of automated static analysis, syntax analysis will just check whether the specification is in accordance with the rules describing correct statements in the particular language. Semantic analysis uncovers the use of variables without declaration, data-type clashes or references to unspecified parts. In general, syntax and semantic analysis assure that the specified system is well-defined, complete and self-consistent from a language point of view. A system that has passed these checks is comparable to a program that has been successfully compiled [11]. However, these techniques are not powerful enough to reveal whether the functionality of a system is correctly specified or not.

Dynamic Analysis

Dynamic analysis techniques execute the model described by the behavioural parts of the formal specification. This allows different approaches of validation. First of all, it is possible to test the executable model like any other executable program. Test scenarios with suites of test cases can be applied to the model. Comparing the expected behaviour with the observed behaviour as a result of a stimulus sent to the system allows validation of the specification on a black box basis.

A white box driven approach is to explore the functional behaviour of the model step by step. During such an exploration, two classes of properties can be checked [11]: General properties that apply to nearly any system independent of the particular requirements. They can be defined without reference to a particular system. Examples of such properties are the absence of deadlocks and livelocks, no reading of variables before assignment, or range violations. The second class of properties is system specific properties. They concern the special dynamic behaviour of the model depending on user requirements. By specifying assertions, invariants or other observable conditions which are particular to the modeled system, a validation tool can check whether these conditions hold during the execution of the model.

While the general properties can be checked automatically, checking the system specific properties requires an explicit description of the properties. Making these property descriptions is usually time consuming and may in turn be error-prone.

Figure 1 shows the relationship between model, formal specification and the corresponding implementation.

Simulation Based Validation

The most common dynamic analysis technique used in practice is based on a simulation-like state space exploration [12]. Another application for simulation is performance validation. This topic concerns non-functional timed properties of the examined system and is not covered here.

Exploring the state space of a model allows a white box based validation of the specification. Starting from the initial state, the executable transitions leading to successor states can be calculated. Analysing each visited state with respect to general and user-defined properties enables us to validate the specified model. Two classes of exploration algorithms have to be distinguished: exhaustive and non-exhaustive exploration algorithms.

Exhaustive validation performs an analysis of the complete model. Thus, definite statements about properties such as the absence of deadlocks can be made. The most common exhaustive exploration approach builds the reachability graph of a system by visiting all reachable states. The visited states in this graph are used during the exploration to avoid multiple visits of states. This ensures the termination of the algorithm. A reachability graph comprises all execution paths the system is able to perform. Deadlocks, livelocks or dead code can be discovered in this way. The state diagram of a small sample system and the corresponding reachability graph with a sample execution path is given in Figure 2. Circles depict states, arrows correspond to transitions.

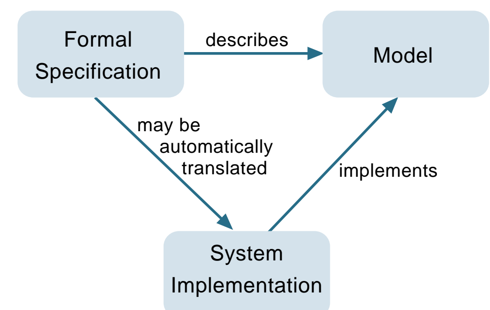

Figure 1 Relationship between model, specification and implementation

model under analysis. Such a guidance is a specialization of the notion of a *validation model* [12]. A validation model is a small, formally specified executable model describing a certain aspect of the system being validated. The complete system specification is checked against the validation model.

In the SDL domain, MSCs and SDL-based observer processes are used to validate a specification with regard to user-defined properties. By giving an MSC that describes some desirable behaviour of the system, the simulator checks for execution paths satisfying the following properties: The execution trace must include all events that exist in the MSC and must not contain any observable event that is not part of the MSC. The sequence of observable events must be consistent with the partial ordering of the events that is defined by the MSC.

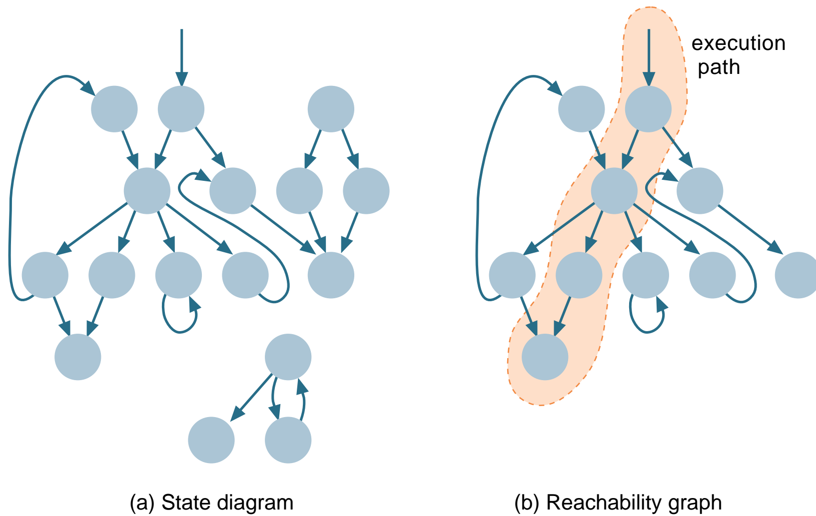

Figure 2 A state diagram and its corresponding reachability graph

Unfortunately, this method is only applicable to small systems. The number of states grows exponentially with the complexity of a system. Non-exhaustive exploration algorithms cope with this state explosion problem by exploring only parts of a system. Certainly, this method cannot prove error-freeness for the whole system, but experience has shown that specification errors manifest themselves in many different states. Therefore, it is not necessary to cover all execution paths of the system to spot errors. The problem is to find a sufficient subset of all paths.

One practical technique is to choose a more or less random subset of depth-bound paths. Most tools provide a *random walk* exploration: Transitions to be executed are chosen randomly, yielding a random exploration of the state space [24]. By repeating random walks, the desired coverage of the specification is obtained.

A more elaborate non-exhaustive technique is the bit-state exploration [12]. Each state is encoded via a hash function to represent an array index. In this way, each state identifies a slot in a hash table used to indicate whether that state has already been visited or not. This enables tools to approximate an exhaustive search of considerably larger systems, because less memory is needed for each state – instead of storing visited states completely, just a one-bit slot per state is needed. Because of the risk of hash value conflicts, bit-state exploration is nevertheless a non-exhaustive validation technique.

Independently of these techniques, the number of states can be reduced by a controlled partial search. SDL tools like *SDT* and *Object*GEODE offer options to let the user limit the state space which has to be validated. For example, validation tools can be guided by *Message Sequence Charts (MSC)* [15] describing scenarios for the

Other Validation Techniques

If a system is specified using a more abstract specification language than SDL, more sophisticated validation techniques can be employed. Such languages are not widely used in the commercial area, because they differ from the popular imperative state machine based specification languages. An example is the μ-calculus which is based on temporal logic.

An approach well studied in the academic area is model checking. The idea behind this is to express states and possible transitions by means of logical predicates. Thus, questions concerning reachability can be answered by automatically solving logical equations. Symbolic model checking is still based on state exploration, but algebraic transformations are used and states are represented symbolically, not explicitly in a reachability graph. These symbolic logical formulas can be transformed into boolean expressions which in turn can be represented very efficiently by binary decision diagrams (BDDs) [3]. This allows us to explore larger systems [4]. The *Xeve/Estérelle* toolkit is an example of a symbolic model checking tool [1]. Another widely used model checker is the *Spin* tool [12]. It uses a partial order reduction technique to cut down the state explosion. In [19], a study of the interconnection of *Object*GEODE with a model checker is presented.

The *Las* tool introduces a new validation approach [7]. Linear programming is used to prove properties on communicating automata yielding a polynomial complexity. Linear programming is an efficient technique to solve certain optimization problems. This approach avoids the construction of the reachable state space of a model. Instead, it calculates directly on the formal model whether or not a certain

property holds by searching an execution path satisfying the property. The practical value of this approach is not yet clear and further studies are necessary.

Testing

Testing is an important aspect of today's product development cycle. The complexity of new telecommunications systems increases constantly; the amount of time needed for testing and its cost grow accordingly. In order to reduce time and cost for testing, research institutes, standardization organizations, tool providers and industry are actively developing formal methods, languages and tools for test generation and test execution. In this section, we will focus on test generation for software systems.

In the domain of software testing, many categories are distinguished: Domain, risk, load, stress, scenario, feature, integration or user testing are just some of the common testing methods. We will concentrate on conformance testing, where the conformance of a system implementation (the *Implementation Under Test (IUT))* with regard to a (formal) system specification is checked. The *Conformance Testing Methodology and Framework (CTMF)* has been standardized with the ISO/IEC 9646 multipart standard [13]; it provides the foundations for the methods and tools discussed here.

Figure 3 shows the relationship between specification, *test suite* and implementation. A test suite consists of a set of test cases. Each test case describes sequences of signals which are exchanged between the tester and the IUT through *Points of Control and Observation (PCO)*. At the end of each sequence, one of the three possible test verdicts *pass*, *inconclusive* and *fail* is assigned. During test execution, the execution of each test case should end with a pass verdict. If this is the case, then a *conformance statement* can be made about the implementation.

There are two main problems which have to be solved in order to get high-quality conformance test suites: First, a set of test cases has to be identified which as a whole guarantees a certain level of conformance. Second, a test suite has to be generated which contains the information necessary to derive executable tests.

Test Case Generation

There are two approaches to test case generation: Automatic (exhaustive) test case generation, and test-purpose-based test case generation.

Exhaustive Test Generation

Exhaustive test generation methods aim at identifying and generating complete test suites automatically. The input is a *Finite State Machine (FSM)* or some variation, e.g. an *Extended FSM (EFSM)*, *Communicating FSM (CFSM)* or *Communicating Extended FSM (CEFSM)*. EFSM is the underlying model of a one-process SDL specification; CEFSMs correspond to SDL specifications with multiple communicating processes.

An important aspect of exhaustive test generation methods is the definition of the test suite goal. One commonly used goal is the establishment of a guaranteed *fault* coverage, e.g. 95 %: If a test suite is executed completely and no error is detected, then the statement can be made that the implementation is guaranteed to be free of 95 % of all possible faults (it may contain any number of the remaining five percent of possible faults, though). There are several types of faults, the main ones being *output* and *transfer* faults [20]. Output faults occur if the output of a transition does not match the expected output; transfer faults occur if a transition ends in the wrong tail state.

Another common test suite goal is to obtain a certain *code* coverage. Taking an SDL specification as an example, a code coverage of 90 % would mean that 90 % of all SDL symbols in the specification are covered by at least one test case. In general, test suites which obtain a high fault coverage are considered to be of higher quality than the ones which rely on code coverage.

Unfortunately, due to the state space explosion problem, current methods for exhaustive test generation can only handle specifications which are very limited in size and often restricted with regard to the specification possibilities offered by languages such as SDL. Nevertheless, methods and tools have been developed which produce test cases to test the components of a CEFSM in isolation or in context (embedded testing); examples can be found in [2] and [5].

Test-Purpose-Based Test Generation

Exhaustive test generation as described in the previous section has two major drawbacks: First, it can only be applied to small systems or parts of systems; it cannot be used for today's real-world complex systems, because of state space explosion and infinite state spaces. The second, less technical but nevertheless important drawback is the lack of test case documentation.

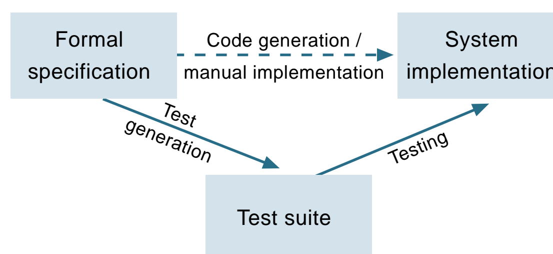

Figure 3 Relationship between specification, test suite and implementation

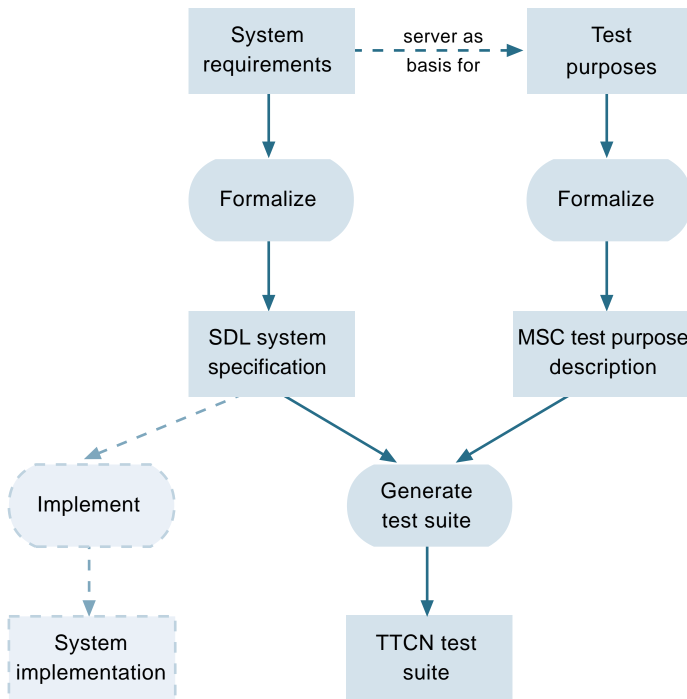

Figure 4 Test-purpose-base test generation

Automatically generated test cases tend to be rather non-descriptive sequences of test events which do not make much sense to the human reader. Test-purpose-based test generation methods alleviate both these problems.

The main idea behind test-purpose-based test generation is the following: Before a system is specified formally and implemented with some programming language, there are usually requirements capture and analysis phases. In these phases, engineers define the important signal flows. Later on, the system is specified with a formal specification language. It is obvious that the specification should exhibit the behavior defined during requirements capture, so the signal flows defined there can now be used as test purpose descriptions. To generate test cases, the system specification can be simulated against the test purpose descriptions. During the simulation run, the signal exchange at predefined *Points of Control and Observation (PCO)* is observed. Signal sequences which correspond to the test purpose description lead to a *pass* verdict; observed signals which are correct according to the specification but which are not expected in the test purpose are marked with an *inconclusive* verdict. This method has originally been proposed in [10] using SDL as the system specification language and MSCs for test purpose description (Figure 4).

Obviously, with the test-purpose-based approach, the quality of a test suite cannot be measured with fault coverage criteria.[1)] The completeness and quality of the test suite mainly depends on the experience of the persons defining the test purposes. However, the generated test cases are guaranteed to be consistent with the formal specification. This consistency is not given by default, since test designers are not necessarily aware of the formal specification. Furthermore, efficient algorithms and tools exist which can generate test cases even for very complex systems [9]. Last but not least, test purpose descriptions can also be used as documentation for the system. The applicability of this method to various real-world systems and protocols has been shown [22], [23].

Test Generation without a Formal System Specification

For many existing systems in industry, formal system specifications have never been developed and it would not be cost-effective to do a specification just for test generation purposes. In other cases, only partial specifications exist. Nevertheless, the idea of being able to specify test purposes with MSCs instead of directly writing test cases in a specialized test language has been widely accepted. For this reason, industry requires tools which are able to translate test purpose descriptions directly into test cases.

Test Suite Generation

The road to executable test suites does not end with the identification and generation of test cases. Test suites must be saved either in a proprietary language defined by the test equipment vendor or in the standardized *Tree and Tabular Combined Notation (TTCN)* [14]. If TTCN is chosen and the test suite is supposed to be easily readable and understandable by humans, then the following optimizations should be done automatically by the test generation tool:

- All declarations should be generated;
- The number of constraints should be minimized by merging identical constraints and by supporting constraints parameterization;
- Constraints should get meaningful names;
- The number of test steps should be minimized through parameterization.

If Concurrent TTCN is used to specify tests for a distributed test architecture, then the tool has to support additional features: Test case descriptions have to be split into descriptions for all test components and synchronization messages should be generated automatically.

[1)] *It is possible to measure the code coverage during simulation, though.*

The ultimate goal of any industrial-strength TTCN test generation tool must be to generate test suites which require no manual postprocessing.

Tools

There are two SDL-based test generation tools which are available commercially: *Autolink* [9] has been developed in a joint project with Telelogic AB by the Institute for Telematics of the University of Lübeck; it is part of the Telelogic Tau tool suite. *TestComposer* [18] has been developed by Verilog as part of the *Object*-GEODE tool set. Both tools are based on the test-purpose-based test generation methodology and they contain many similarities. Below, we just give a summary of the distinguishing features of the tools.

Autolink has been available since 1997 and it has been evaluated and used in projects of the European Telecommunications Standards Institute (ETSI), as well as telecommunications companies. Because of the incorporation of many features requested by users, its strengths lie in the readability of generated test suites and in its adaption to industry realities. Therefore, besides offering state exploration based test generation, Autolink also supports the direct translation of MSCs into test cases without the need of a complete formal system specification. Other unique features are:

- Generation of Concurrent TTCN output including the automatic generation of coordination messages;
- Inclusion of a simple configuration language which allows to define rules for automatic constraints naming, constraints parameterization and test suite variables (PIXIT);
- Support for distributed test generation.

Although TestComposer has been developed from scratch, it is based on a relatively long history of research and tools in the domain of automatic test generation. Its strengths lie in its state space exploration techniques. For example, TestComposer is able to generate test cases from partially defined test purpose descriptions; the missing parts are filled in automatically. Other distinguishing features are:

- Automatic postamble[2)] computation;
- Comprehensive timer support;
- Output of test suites in a user-definable format.

Conclusions

Although it may not seem obvious at first sight, there are several similarities between validation and automatic test case generation. Both techniques require searching the state space of the system under investigation. During validation, the tools look for peculiarities in the state space such as unspecified reception or deadlocks. During exhaustive test case generation, the whole state space is searched for those test cases which can detect faults in an implementation. In the test purpose based method, the state space is explored to check if a trace exists which matches a predefined and formally specified test purpose. This way, test cases are generated which lead to pass and inconclusive test execution verdicts. Due to these similarities, tools such as Autolink [9] and TestComposer [18] make use of techniques originally developed for validation, such as described in [12].

Validation and test case generation both suffer from the state space explosion problem which makes it impossible to exhaustively validate or test systems of practical size. However, the use of available tools is already beneficial and highly recommendable: For validation, advanced state space exploration algorithms have been developed which allow to explore significant parts of the state space of real-world system specifications in a feasible amount of time. For test generation, support of the pragmatic approach of using (human specified) test purposes combined with state space exploration has made tool-assisted development of high-quality test suites a reality.

Experience in industry and ETSI has shown that the initial effort to develop a formal specification can be quite high. However, this effort is offset by the ability to detect design errors at an early development stage through validation, by the possibility of automatic code generation and the ability to easily develop test suites through automatic test generation. All in all, relevant reductions in time-to-market and development cost can be expected through the use of formal techniques.

A lot of research is going on in the field of formal specification, validation and test generation. At the end of 1999, new versions of SDL and MSC have been standardized by the *International Telecommunication Union (ITU)*. With the *Unified Modeling Language (UML)*, a new notation for object-oriented software development has been standardized by the *Object Management Group (OMG)*. At ETSI, guidelines are developed on how to use object-orientation in the standardization and specification process of telecommunication systems. Also at ETSI, a

---

[2)] *A postamble is a sequence of test events to bring the IUT into a stable, well-defined state.*

completely new version of the testing notation TTCN is in the final stages of development. Meanwhile, research institutes and tool providers continue to develop enhanced methods and tools for validation and test generation.

References

1 Bouali, A. *Xeve : an estérel verification environment.* Sophia Antipolis, Inria, 1997. (Technical Report 0214.)

2 Bourhfir, C et al. A test case generation tool for conformance testing of SDL systems. In: Dssouli, E et al. [8], 405–419.

3 Bryant, R E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers,* 35 (8), 667–691, 1986.

4 Burch, J R et al. Symbolic model checking : 10^{20} states and beyond. In: *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science,* Philadelphia. Los Alamitos, IEEE Computer Society Press, 1990, 428–439.

5 Cavalli, A et al. Hit-or-jump : An algorithm for embedded testing with applications to in services. In: *Proceedings of the Joint International Conference FORTE/PSTV'99. IFIP TC6 WG6.1.* Boston, Kluwer, 1999.

6 Cavalli, A, Sarma, A (eds.). SDL'97 – time for Testing. *Proceedings of the Eighth SDL Forum,* Evry, France, September 1997. Amsterdam, Elsevier, 1997.

7 Devulder, S. A comparison of lpv with other validation methods. In: *Proceedings of ASE-99: The 14th IEEE Conference on Automated Software Engineering,* Cocoa Beach. Los Alamitos, IEEE computer Society Press, 1999.

8 Dssouli, R, Bochmann, G v, Lahav, Y (eds.). *SDL'99 – the Next Millennium, Montréal. Proceedings of the Ninth SDL Forum.* Amsterdam, Elsevier, 1999.

9 Ek, A et al. Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. In: Cavalli and Sarma [6], 245–259.

10 Grabowski, J, Hogrefe, D, Nahm, R. Test case generation with test purpose specification by MSCs. In: *SDL'93: Using Objects. Proceedings of the Sixth SDL Forum*. Amsterdam, North-Holland, 253–265.

11 Hogrefe, D. Validation of SDL systems. *Computer Networks and ISDN Systems,* 28 (12), 1659–1667, 1996.

12 Holzmann, G. *Design and Validation of Computer Protocols.* Englewood Cliffs, Prentice-Hall, 1991.

13 ISO/IEC. *Information technology – Open systems Interconnection – conformance testing methodology and framework*. 1994. (International ISO/IEC multipart standard No. 9646.)

14 ISO/IEC. *Information technology – Open systems Interconnection – conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN).* 1997. (International ISO/IEC standard No.9646-3.)

15 ITU-T. *Message Sequence Charts.* Geneva, 1999. (ITU-T Recommendation Z.120.)

16 ITU-T. *Specification and Description Language (SDL)*. Geneva, 1999. (ITU-T Recommendation Z.100.)

17 Sarma, A, Ellsberger, J, Hogrefe, D. *SDL – Formal object-oriented language for communication systems.* London, Prentice-Hall, 1997.

18 Kerbrat, A, Jéron, T, Groz, R. Automated test generation from SDL specifications. In: Dssouli et al. [8], pages 135–151. Proceedings of the Ninth SDL Forum.

19 Kerbrat, A, Rodriguez-Salazar, C, Leujeune, Y. Interconnecting the objectgeode and caesar-aldeberan toolsets. In: Cavalli and Sarma [6], 475–490.

20 Petrenko, A, Bochmann, G v, Yao, M. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems,* 29 (1), 81–106, 1996.

21 Rakitin, S. *Software Verification and Validation.* Norwood, Artech House, 1997.

22 Scheurer, R. *Demonstrating the Applicability of Automatic Test Case Generation Methods.* PhD thesis. Berne, University of Berne, 1997.

23 Schmitt, M et al. Autolink – putting SDL-based test generation into practice. In: *Proceedings of the 11th International Workshop on Testing of Communicating Systems (IWTCS'98),* Tomsk, Russia, 1998. IFIP TC6. Amsterdam, Kluwer, 1998, 227–243.

24 West, C. Protocol validation. *Computer Networks and ISDN Systems,* 24, 1992.