Automatic Test Generation Based on Formal Specifications

Practical Procedures for Efficient State Space Exploration and Improved Representation of Test Cases

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Michael Schmitt
aus Orsoy

Göttingen 2003

D 7
Referent: Prof. Dr. Dieter Hogrefe
Korreferent: Prof. Dr. Jens Grabowski
Tag der mündlichen Prüfung: 3. April 2003

Acknowledgments

I would like to thank my supervisor Prof. Dr. Hogrefe for his assistance and the possibility to research under perfect conditions. The work at the Institute for Telematics in Lübeck has been a very positive experience. I appreciate that I was allowed to act independently and to take on responsibility in a trustful atmosphere.

I would also like to thank my former colleagues in Lübeck who have accompanied me through the years. Special thanks go to Jens Grabowski and Beat Koch with whom I worked together in the AUTOLINK project and who shared endless discussions and debates on testing and test generation with me.

This thesis would not be in its current state without the numerous proof-readers who spent their spare time on finding typing errors, missing indexes, etc. and who gave helpful hints at how to improve my dissertation. I thank Zhen Ru Dai, Michael Ebner, Andreas Heuer, Maeve Hölscher, Helmut Neukirchen, Cornelia Rieckhoff, and Uwe Roth for their efforts.

Another major contribution to this dissertation was made by my parents who not only conceived me but also encouraged me to study computer science.

This thesis is dedicated to Susanne. Thanks to her support and care, this dissertation has become reality. I hope that I will ever be able to return what she has done for me.

Contents

List of Figures

List of Algorithms

1 Introduction

Modern telecommunication systems involve complex interactions between distributed components. Very often, these systems are heterogeneous and their components come from many different vendors. To ensure that products of different companies can interact with each other, cross-national organizations such as the *European Telecommunications Standards Institute* (*ETSI*) or the *International Telecommunications Union* (*ITU*) define international standards for protocols.

In the past, these standards were specified in natural language. But since informal descriptions can be misinterpreted, formal description techniques are applied in many standards nowadays. Due to their formal semantics, formal description languages allow for precise, unambiguous specifications. Two specification languages that are used in the telecommunication area are the *Specification and Description Language* (*SDL*) and *Message Sequence Chart* (*MSC*). While SDL is used for the specification of complete systems, MSC allows to describe single scenarios.

In recent decades, computer science has made great efforts to improve the quality of software. Nevertheless, a software product has to undergo extensive tests before it can be used in practice. The process of determining the extent to which an implementation fulfills the requirements of its specifications is called *conformance testing*.

Thorough testing is expensive and time-consuming. On the other hand, testing is always incomplete; it can detect errors in the implementation but it can never prove their absence. For economical reasons, testing calls for a systematic and efficient approach. In particular, this goes for the initial phase of conformance testing in which a suitable set of test cases must be defined for a given protocol. Typically, the standardization organizations take over this task and provide the manufacturers with test suites that are specified in a standardized test language such as the *Tree and Tabular Combined Notation 2* (*TTCN-2*).

If a formal specification is available, test cases can be derived automatically from the specification. This can be achieved, e.g., by means of simulation. Automatic test generation based on formal specification leads to a faster, cheaper, and less error-prone testing process. It ensures that the generated test cases are correct with regard to the specification. Moreover, their effectiveness can be assessed and quantified.

This thesis deals with the automatic generation of test suites for conformance testing. In particular, work has been done in the following problem areas:

1. Test case generation for test architectures where the tester itself is a distributed system.

2. Efficient exploration of the state space of the specification that retrieves test cases with a high coverage in a reasonable time.
3. Improved readability of generated test suites by means of user-defined rules and automatic structuring of data descriptions.

The framework in which all these solutions are embedded is formed by the AUTOLINK project. During this five-year joint project with TELELOGIC SA, Malmö, the Institute for Telematics (University of Lübeck) developed a commercial tool that allows to generate TTCN-2 test suites based on SDL system specifications and MSC test purposes. Various projects at ETSI have proved the successful application of AUTOLINK.

Beside the AUTOLINK project, two other works in the field of testing and test generation are considered in this thesis: In 2001, a new universal testing language, called *Testing and Test Control Notation 3* (*TTCN-3*), was released. The author was involved in its standardization and developed the first free TTCN-3 syntax checker. Experience with AUTOLINK has unveiled that the traditional way of simulating a specification raises some problems that could be solved by symbolic execution. For proof of concept, a prototype has been developed that demonstrates the benefits of symbolic execution.

The Structure of the Thesis

This thesis consists of eleven chapters. The dependencies between the individual chapters are shown in figure 1.1.

In chapters 2 and 3, the fundamental aspects of testing are outlined. Chapter 2, "Foundations of Testing", provides an overview of types of testing, test architectures, and test methods. The *ISO/OSI Conformance Testing Methodology and Framework* is especially emphasized. In chapter 3, "Test Languages", the *Tree and Tabular Combined Notation 2* and its successor, the *Testing and Test Control Notation 3*, are described and compared.

The theoretical foundations of automatic test generation are explained in chapter 4, "Test Generation Based on Formal Specifications". It summarizes the main concepts of the *ITU-T Framework on Formal Methods in Conformance Testing* and presents different test generation methods. In chapter 5, "High-Level Specification Languages", the formal description techniques *Message Sequence Chart* and *Specification and Description Language* are introduced. They are used for automatic test generation in this thesis.

The AUTOLINK project is described in chapters 6 to 9. Chapter 6, "The AUTOLINK Tool", provides an overview of the test generation process. In addition, the interpretation of MSCs for test generation purposes is considered and two case studies are introduced. In the following three chapters, solutions for specific test generation problems are presented. Chapter 7, "Test Generation for Distributed Test Architectures", deals with methods for specifying synchronization among different test components and presents an algorithm for the generation of test cases for concurrent TTCN-2. Chapter 8, "The TREE WALK Search Strategy", describes a new deterministic search strategy that achieves a higher system coverage than traditional approaches and produces test cases without redundant

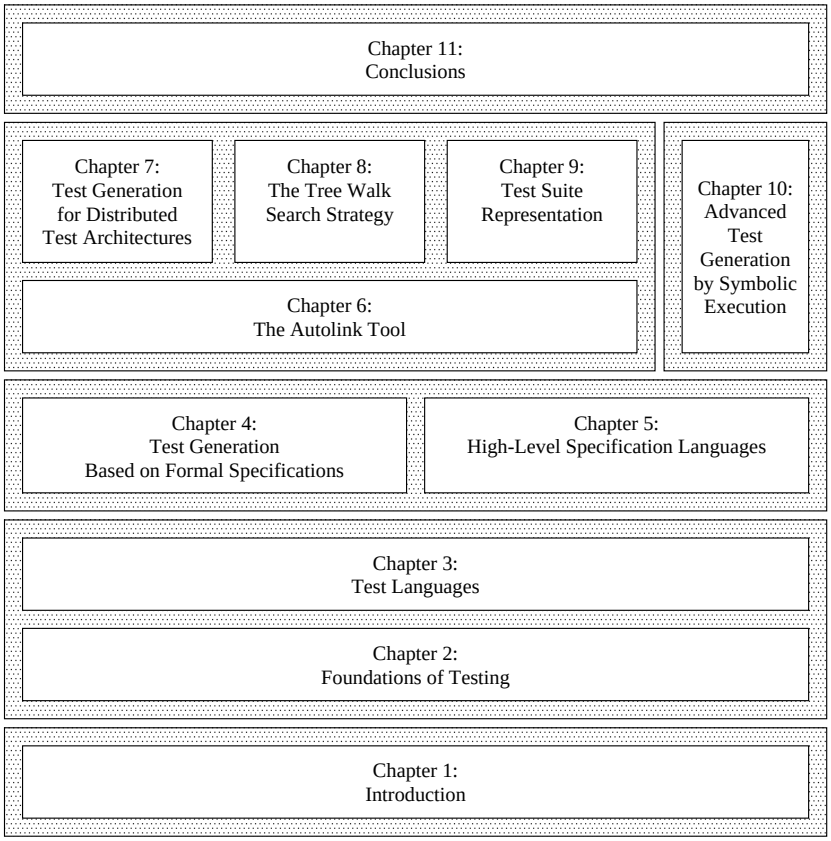


Figure 1.1: Structure of the thesis

test events. Chapter 9, "Test Suite Representation", is concerned with the readability of automatically generated test suites. Two solutions are proposed: Customization by user-defined rules and automatic constraint structuring with no or only minor intervention by the test specifier.

One of the greatest challenges in automatic test generation is the modeling of the environment when simulating a specification. In chapter 10, "Advanced Test Generation by Symbolic Execution", a prototype is presented that demonstrates how to cope with this problem.

Each of the previous chapters concludes with one or more case studies that are based on three protocols, namely *Inres*, *Core INAP CS-2*, and *VB5.2 BBCC*. In addition, possible improvements are discussed, where appropriate. In chapter 11, "Conclusions", some ideas on the application and future perspective of automatic test generation in general are presented.

The document is completed by three appendices: In appendix A, the abbreviations used throughout this thesis are listed. Appendices B and C include complete TTCN-2 and TTCN-3 test suites for the Inres protocol of which parts are presented in chapter 3.

2 Foundations of Testing

Testing is a method or process that aims at detecting errors in a system or a system component. Beside verifying and analyzing methods, testing is one of the three analytical ways of ensuring the quality of a product in terms of functionality and reliability (Liggesmeyer, 1990, p. 28). But since verification mostly fails due to the complexity of modern systems, and static analyses of their structure and complexity (based on metrics and control/data flow) can only give hints at potential problems, testing is often the only practicable method to create confidence in a system.

Testing represents an extensive and multifaceted process within the software and hardware life cycle. A classification and description of different types of testing is given in section 2.1. This thesis is concerned with conformance testing whose purpose is to determine the extend to which an implementation complies with its specification(s). The core concepts of conformance testing are presented in section 2.2. A short survey of the testing process and the persons involved is given in section 2.3. Finally, section 2.4 discusses the major aspects of abstract test methods and test configurations.

2.1 Classification of Tests

Testing is a generic term that subsumes many different methods, procedures, and objectives of various application areas (see, e.g., Peng and Wallace (1993); Walter et al. (1998)). Accordingly, there are several possible ways to define and classify types of tests (see, e.g., Balzert (1998, chapter 5) and Coward and Ince (1995, chapter 2)).

A first major distinction can be made between static and dynamic test methods:

Static methods are based on the analysis of code. Manual test methods that fall into this category are *review* (software requirements, design, and code are presented to an audience for comments and approval), *inspection* (the work is examined by a person or group other than the author), and *structured walkthrough* (the developer guides his colleagues through a segment of design or code).

Contrarily, *dynamic test methods* are based on the execution of the system to be tested. Typically, the system is stimulated by some input and the resulting output (response) is observed and appraised.

In the following, dynamic testing is characterized by five properties:

- The type of implementation
- The objective of the test

- The knowledge of the internal structure of the system
- The authority used for determining the test results
- The selection of the test data

Type of Implementation. According to whether the system to be tested is a physical device or a piece of software, two types of testing can be distinguished:

- **Hardware testing:** The system is tested on the level of single transistors, circuits, or functional entities such as CPU caches or video graphics cards. The exchange of test data between the tester and the system takes place on the physical layer where the inputs and outputs are either analog signals or discrete data.
- **Software testing:** The system represents a program whose behavior is checked by exchanging test data on the logical, i.e., non-physical, layer. Although a program cannot exist without an underlying physical machine, software testing simply assumes its correctness and abstracts from it.

Both types of testing also occur in combination, e.g., if a hardware device with embedded software, such as a medical diagnosis system, is tested.

Software testing itself applies to all kinds of programs (and their specific interfaces) including communication protocol instances (service access points), libraries (application programming interfaces), and user applications (graphical user interfaces). Tests can be applied to single methods and functions, classes and methods, or complete systems.

Test Objective. There are different types of requirements that an implementation must fulfill to work correctly. Accordingly, there are various kinds of test objectives:

- **Functional tests** check the functional behavior of a system, i.e., its correct input/output behavior in certain states of program execution. Passing functional tests successfully is an essential prerequisite for any implementation.[1]
- **Real-time tests** are made to test the correct behavior of a system with regard to its real-time requirements. One can distinguish between two types of requirements: *Hard real-time requirements* demand that a system responds to some input within a fixed period of time (e.g., in flight control systems). Contrarily, *soft real-time requirements*, as imposed on multimedia systems, allow for some variance in the response times. Their compliance is examined by statistical measures (→ **Quality of Service tests**).
- **Performance tests** analyze the system with regard to response times, CPU usage, memory requirements, or other quantifiable features in normal and overload situations. Performance data can either be obtained externally (e.g., by measuring the system's process time relative to the volume of input data) or by instrumentation of the code (e.g., when measuring the average execution time of a single function).

[1] Please note that in literature, *functional testing* is also used as synonym for *black box testing*.

- **Load tests** are used to check the behavior of a system under heavy load. Their purpose is to ensure that the system is capable to respond in time, even if the test data are sent to the system at a high rate.
- **Stress tests** analyze the behavior of a system under unusual conditions. Exceptional states are caused, e.g., by inopportune or malformed test data that are fed into the system at a high rate.
- **Integration tests** make sure that several software and/or hardware elements are combined in such a way that all intermodule communication links are established correctly.
- **Portability tests** aim at demonstrating that a software system can be ported to another hardware or software platform.
- **Penetration tests** analyze the system with regard to vulnerabilities that allow a potential attacker to gain unauthorized rights or even to take control of the system. For that purpose, the system is checked for erroneous configurations and known software bugs. Typically, the system is a server that is running on a host connected to a network.
- **Usability tests** aim at assessing the human-computer interface of a system with regard to criteria such as adequacy, simplicity, clarity, and consistency. Typically, usability tests are performed by recording and evaluating the behavior of an external test person interacting with the system (gestures, response times, eye movement, etc.).

System Knowledge. The amount of information given about the system implementation determines the way in which tests can be specified. Three types of testing can be distinguished:

- **Black box testing:** The system is considered a component with interfaces to its environment that is supposed to fulfill a specific task but whose internal structure is unknown. Black box tests are derived from the specification which the system is based on as this is the only source of information.
- **Gray box testing:** Some information about the workings of the system are known in advance or observable during test execution (e.g., the number of states or the current state in case of state-machine like implementation). This knowledge can be used when specifying test cases.
- **White box testing (Glass box/Structural testing):** All internal details of the system, typically in form of its program code, are known. The set of appropriate tests (test data) is determined by a preceding control flow or data flow analysis.

Please note that in literature, the term *white box testing* is sometimes used to denote tests during which the internal sequence of events can be inspected by means of monitoring.

Authority for Test Results. Another criterion for classifying tests is the "authority" which is used as reference to decide whether a system passes a test successfully.

- **Conformance testing** is used to determine the extend to which an implementation conforms to its specification. In general, no information about the internal structure of the implementation is given, i.e., conformance testing implies black box testing.
- **Interoperability (compatibility) testing** is a means to ensure that an implementation, e.g., a protocol entity, is able to work with other implementations. There are two kinds of interoperability testing: **Active testing** allows to insert selective errors in the communication between two systems and monitor their reactions, whereas **passive testing** is restricted to testing valid behavior only.

  Even if two implementations that claim to conform to the same specification are able to interoperate, it does not necessarily mean that they in fact conform to the specification. The other way round, two systems conforming to the same specification do not necessarily interoperate if, e.g., the specification is ambiguous.[2]
- **Regression testing** is repeated whenever significant code changes are made. Selective retesting of a product ensures that modifications do not have unintended side effects and systems requirements are still met. Since former software releases are assumed to fulfill the system requirements, they are supposed to pass all tests and can be considered as reference. Thus, only deviations in the test outcome between the latest release and its predecessors must be considered. (Testing of new features is called **progressive testing**.)
- **Back-to-Back/Comparison testing** is a process by which the output of two or more systems that are based on the same specification is compared to detect anomalies. Comparison testing does not detect specific errors but only discrepancies. If all implementations contain the same error, it will not be discovered.

Test Data Selection. During test execution, the system is stimulated with test data and its response is compared with the set of valid outputs. The extend to which test data are considered from the input domain of the system determines the comprehensiveness of the tests.

- **Exhaustive testing:** Any possible test data is used for every possible point of execution. In practice, the domain of most inputs is rather large and the response of (stateful) systems may also depend on former inputs. Thus, exhaustive tests can only be made for systems with very small complexity.
- **Partition testing:** The input domain is divided into several disjoint subdomains from which only one test data is chosen. These subdomains may be determined by a data flow analysis of the systems or by some general rule. For instance, a negative value, zero, and a positive value might be chosen for each integer.

[2] The fact that a specification can be interpreted in different ways may even remain unnoticed until interoperability tests are performed.

- **Boundary value testing:** As a special kind of partition testing, test data are selected on or around range limits or boundaries. In addition, test data should be chosen to force the output to its extreme values (or beyond the specification boundaries). For integers, the value zero often causes problems (e.g., divisions by zero). For sequences of data, attention should be focused on their first and last element and on empty lists.
- **Random testing:** Test data are selected in a random manner.
- **Mutation testing:** Based on the system specification or the program itself, a large set of modified versions, *mutations*, are created that differ from the original program by a single characteristic such as a missing or modified statement. Test data are selected in such a way that the test result of at least one test case differs between each mutant and the original program.

2.2 Conformance Testing Concepts

In the following, the major concepts of conformance testing as defined in ISO/IEC International Standard 9646 (ISO/IEC, 1994b) are introduced.[3] ISO/IEC IS 9646 is a seven-part standard that is better known as *Information technology – Open systems interconnection – Conformance testing methodology and framework* (*CTMF*). Although it focuses on OSI protocol testing, most of its concepts apply to conformance testing in general and even other test methods.

2.2.1 Conformance Requirements

The purpose of conformance testing is to find out whether an implementation conforms to its base specification(s). In order to do so, an implementation must fulfill both static and dynamic conformance requirements.

Static conformance requirements specify the minimum set of capabilities which have to be implemented to facilitate interworking. In addition, they define limitations of the combination of capabilities. In this context, a *capability* denotes a set of functions defined in the protocol specification that is supported by the implementation.

Dynamic conformance requirements specify the observable behavior (in terms of communication) that is permitted according to the base specification(s).

Within the presented framework, conformance testing means functional black box testing. Therefore, while violations of conformance requirements might be detected during test execution, their absence cannot be guaranteed.

[3] An alternative framework for conformance testing is defined by ISO/IEC International Standard 13210 (ISO/IEC, 1994a). It specifies the general requirements and test methods for measuring conformance to *POSIX* (*Portable Operating System Interface for UNIX*) standards.

2.2.2 Test Cases

A *test case*[4] is a specification of all actions that need to be performed to achieve a specific *test purpose*. Ideally, a test case should focus on a single or a small set of related conformance requirements.

IS 9646-1 (ISO/IEC, 1994b, pp. 14–15) defines four types of tests which vary in the extend to which they give evidence for conformance:

- **Basic interconnection tests** are used to ensure that the main features of the specification are implemented correctly and no severe case of non-conformance exists.
- **Capability tests** check that the observable capabilities of the system conform to (a) the allowed combination of capabilities as stated in the *static conformance requirements* and (b) the capabilities listed by the supplier in his *implementation conformance statements* (*ICSs*).
- **Behavior tests** aim at testing the implementation thoroughly, covering the full range of specification requirements (within the capabilities of the implementation). Behavior tests represent the majority of all conformance tests.
- **Conformance resolution tests** provide an in-depth analysis of the behavior of a system with regard to particular conformance requirements. Typically, they are non-standardizable as their execution involves system-specific diagnostic facilities.

ISO/IEC IS 9646 distinguishes between two types of test cases that are related to each other: An *abstract test case* is specified according to an (*abstract*) *test method* (see section 2.4) which describes how an implementation is to be tested. However, the specification is made on a level that abstracts from the concrete equipment and procedures, i.e., the *means of testing* (*MOT*). Thus, for test execution, an *executable test case* must be derived from an abstract one.

A test case must be defined in such a way that the *system under test* (*SUT*, see 2.4) starts and ends in a *stable testing state* which is maintained sufficiently long by the SUT – without further input from the tester – to bridge the gap between the execution of two test cases. Typically, the stable testing state is identical to an *idle testing state*, i.e., a state in which there are no open connections and the state of the SUT is independent from test cases executed previously. This guarantees that all test cases can be executed in isolation and their execution order does not have any impact on the test result.

A test case consists of sequences of atomic *test events* such as sending or receiving a message. A *valid test event* is a test event that is syntactically and semantically correct and occurs when allowed to do so by the specification. An *invalid test event* is a test event that violates at least one of these conformance requirements. An *inopportune test event* is an invalid test event that occurs when not allowed to do so according to the specification.

[4] For convenience, the term *test* is used as a synonym for *test case* throughout this thesis.

Each sequence of test events stands for a *foreseen test outcome*. Conceptionally, a test case can be structured into three parts:

- The *test preamble* comprises the sequences of test events from the starting stable testing state to an initial testing state.
- The *test body* contains the sequences of test events that achieve the test purpose.
- The *test postamble* comprises all sequences of test events from the end of the test body to the final stable testing state.

Test preamble and test postamble are optional elements whose only purpose is to drive the SUT into the desired states, whereas the real conformance test is made when execution the test body.

2.2.3 Test Verdicts

To each foreseen test outcome, an abstract test case must assign a *test verdict*. ISO/IEC IS 9646 defines three types of verdicts:

- A *pass verdict* indicates that no invalid test event has occurred and the test outcome gives evidence that the implementation conforms to its specification(s).
- A *fail verdict* indicates that at least one invalid test event has occurred or that the observed test outcome proves non-conformance of the implementation to its specification(s).
- An *inconclusive verdict* is assigned if neither a pass nor fail verdict can be assigned indisputable. This is the case if the implementation acts conforming to its specification(s) but no statement can be made about the particular conformance requirement(s) which are considered by the test purpose.

A *preliminary test verdict* may be assigned at the end of the test body to indicate that the test purpose has been achieved. A *final test verdict* is assigned at the end of the test case.

If, during test execution, a fault is detected in the test case itself, a *test case error* is reported instead of one of the test verdicts above.

2.2.4 Test Suites

A *test suite* is a collection of test cases that are used to perform conformance tests. In addition, it contains general information such as the specification(s) and the test method on which the test suite is based, and statements about its test coverage.

Test suites have a nested structure: Related test cases whose test purposes aim at a common objective can be combined in a hierarchy of (named) *test groups*. Test events within a test case that form a logical unit, such as a test preamble or postamble, can be described in form of a *test step*. It can be reused for the description of other test cases. Test steps are allowed to refer to other test steps.

In analogy to test cases, ISO/IEC IS 9646 distinguishes between *abstract test suites* (*ATSs*) and *executable test suites* (*ETSs*). Throughout this thesis, the terms *test suite* and *test case* are used to denote abstract entities.

2.3 The Conformance Testing Process

Conformance testing comprises several steps that involve single persons or teams with different roles:

- A *client* of a test laboratory submits a system or implementation for conformance testing.
- A *test specifier* develops the abstract test suite and associated documents.
- A *test realizer* provides the means of testing that are required for the test operation.
- A *test laboratory* carries out the conformance tests.

In the following, the phases *test suite development*, *test preparation*, *test operation*, and *test evaluation* are described briefly.

2.3.1 Test Suite Development

The initial task of testing is to define a reasonable set of test cases. As mentioned in section 2.1, test cases for black-box testing must be derived from the specification(s) which the implementation is based on. Normally, specifications are available in a textual and informal manner. In these cases, test cases must be defined manually which bares the risk that the tests itself contain errors, e.g., due to misinterpretation of the standard. For that reason, conformance tests and system implementations should be developed by different teams. If, on the other hand, a specification includes a formal description that forms a normative part, automatic test generation can be applied. The main part of this thesis deals with methods on how this can be achieved.

In the telecommunication area, specifications are defined by international standardization organizations such as the *International Telecommunication Union* (*ITU*), the *European Telecommunication Standards Institute* (*ETSI*), the *Institute of Electrical and Electronics Engineers* (*IEEE*), or the *Internet Engineering Task Force* (*IETF*). In order to take the burden of developing their own set of test cases from the manufacturers, these standardization organizations often publish their protocol specifications along with test specifications in form of abstract test suites.

2.3.2 Test Preparation

In the test preparation phase, the supplier or implementor has to provide all information that is necessary for a (possibly independent) test laboratory to perform the conformance tests: All relevant specifications to which conformance is claimed are identified

in a *system conformance statement* (*SCS*). The capabilities that have been realized for each specification are listed in *implementation conformance statements* (*ICSs*). Finally, additional information about the implementation and its test environment are given in *implementation extra information for testing (IXIT)* statements.

Based on the created document, a suitable abstract test method and a corresponding abstract test suite is chosen. Thereafter, the test laboratory sets up the SUT and the means of testing accordingly. Protocol testing often requires tailored tester hardware to communicate with the SUT. For instance, PCs with special interface devices might be used. In addition, the tester software, ideally based on a real-time operation system, needs to be configured. If a test suite is given in a standardized test language like the ones presented in chapter 3, a compiler or interpreter is able to translate large parts of an abstract test suite into an executable one. Equipment-specific extensions such as synchronization between different tester components which an ATS abstracts from are implemented in a run-time environment.

2.3.3 Test Operation

The test operation phase starts with a *static conformance review*. The ICSs provided by the supplier are analyzed respecting the *static conformance requirements* of the standards listed in the SCS. The purpose of this review is to detect invalid combinations of implemented capabilities based on the statements of the supplier. In addition, the IXIT documents are checked for consistency.

To check the *dynamic conformance requirements*, the parameters of the test suite(s) are set according to the declarations in the ICSs and IXITs and the set of applicable test cases is determined. Afterwards, a *test campaign* is performed for each parameterized executable test suite. During each test campaign, the observed sequences of test events as well as other information about the test execution, e.g., test verdicts, are recorded in a *conformance log*.

2.3.4 Test Evaluation

After the test operation, the test results are determined based on the conformance logs produced during the test campaigns. In case of uncertainty, the correctness of test verdicts for individual test cases can be checked due to the complete logging of the test outcome.

In a final step, *conformance test reports* are created based on the results of the static conformance review and the test campaigns. They give a summary of the actual conformance of the implementation to its specification(s) as well as a detailed list of those abstract test cases for which executable test cases were executed, together with their resulting test verdicts.

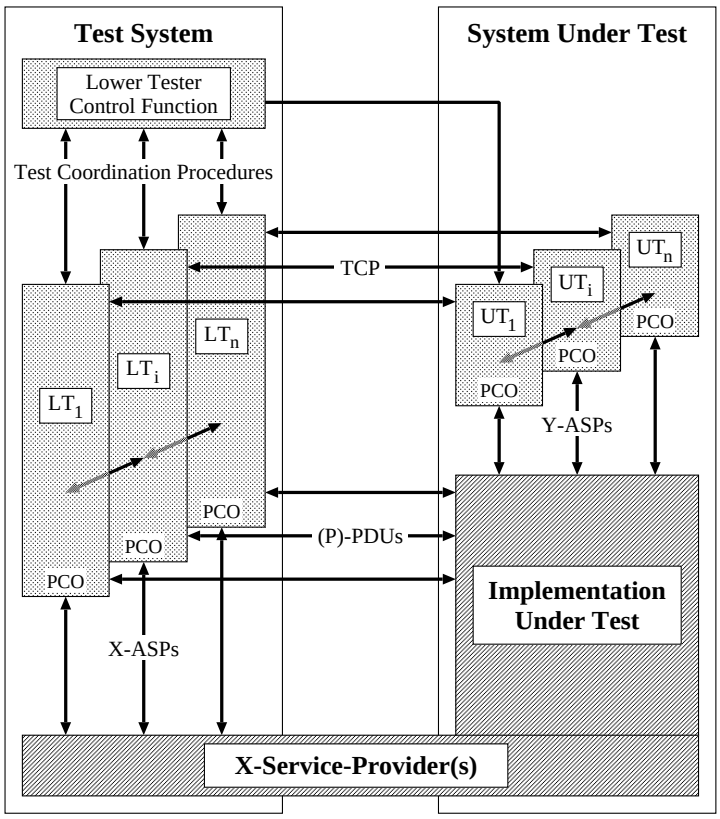


Figure 2.1: The distributed test method in a multi party context

2.4 Test Methods and Configurations

An important prerequisite for specifying an abstract test suite is to define an adequate *abstract test method* that considers the special properties of the specification and the way it is intended to be implemented. To some extend, the chosen test method also determines the concrete *test configuration* that is established during test execution.

ISO/IEC IS 9646 (part 1, p. 25) defines four main abstract test methods, called *local*, *distributed*, *coordinated*, and *remote test method*, and several variants for embedded testing and multi user/multi party testing. These test methods vary in the degree of controllability and observability of test events sent and received by the implementation. To illustrate this, the conceptional view of the distributed test method in a multi party context is presented in figure 2.1.

In each test method, four major entities are identified:

- A *system under test* (SUT), i.e., a real open system provided by the client.
- An *implementation under test* (*IUT*), i.e., the part of the SUT which is subject to conformance testing.

- A *test system* (also called *tester*), i.e., a real system that is provided by the test laboratory.
- A *service provider* which is used for communication between the tester and the SUT.

The SUT may be identical to the IUT or contain additional components. For instance, if the upper boundary of the IUT is a software interface, facilities to control and observe the IUT must be provided within the SUT, since the tester is not able to address the interface directly. This circumstance is reflected by the distributed test method (figure 2.1).

The actual test execution is performed by the following functional entities:

- An *upper tester* (*UT*) controls and observes the upper service boundary of the IUT.
- A *lower tester* (*LT*) controls and observes the lower service boundary of the IUT via an underlying service-provider.
- A *lower tester control function* (*LTCF*) coordinates the lower testers and determines the final test verdict in a multi-party testing context.

In a concrete test configuration, the functional entities listed above are mapped to one or more *test components*, i.e., active elements which are executed in parallel. Each test component realizes the functionality of one or more UTs and LTs. The lower tester control function is integrated in the *main test component* (*MTC*). The number of test components depends on the degree of concurrency in the behavior of the tester. In a multi party context, one test component should be defined for each party. However, it should be noted that, in principle, several test components again may be executed on the same physical device.

The interaction of the tester with the IUT can take place at different *points of control and observation* (*PCOs*). Each PCO is modeled by two queues that contain the test events to be sent to and received from the IUT. Upper testers exchange *abstract service primitives* (*ASPs*) with the IUT; lower testers exchange *protocol data units* (*PDUs*) with the IUT. In practice, these PDUs are encapsulated into ASPs of a service provider which connects the LT with the IUT.

Synchronization among the UTs, LTs, and the LTCF is achieved by *test coordination procedures* (*TCPs*). Their requirements shall be specified for each ATS – either explicitly or implicitly by the test language that is used for the definition of the ATS. On the other hand, the technical realization of test coordination procedures is not prescribed by a test method.

3 Test Languages

The formal specification of tests calls for standardized languages that are widely accepted. In principle, existing programming or script languages could be used for these purposes (see, e.g., the DejaGNU GNU Testing Framework; Savoye, 2001). However, these languages do not provide the appropriate level of abstraction. For instance, traditional programming languages make clear assumptions on low-level implementation aspects such as memory management. On the other hand, high-level testing concepts like test components are not provided and must be added afterwards *on top* of these languages (e.g., by supplementary classes). Moreover, the control structures of languages like C or Java do not fit to the requirements of testing concurrent systems and lead to large and confusing specifications. Script languages have been designed but they are either tailored to a specific test equipment (Moesch, 2001), or to a very restricted application area (see, e.g., the Nessus attack scripting language, Deraison (2000)).

For the reasons given above, the *Tree and Tabular Combined Notation* was developed and published by the *International Organization for Standards* (*ISO*) in 1992. It has gained wide acceptance during the last decade as the primary language for testing of telecommunication protocols. In 1997, the second edition of TTCN (*TTCN-2*) was released. It is described in section 3.1 and used as the target language for automatic test generation within the scope of this dissertation.

In recent years, it has become apparent that the strong relationship between TTCN-2 and the OSI conformance testing methodology imposes restrictions on the application of the language. Therefore, it has been decided to develop a new test language that supports a wider spectrum of testing types and infrastructures (e.g., the *Common Object Request Broker Architecture*, or *CORBA* for short). As a result, the *Testing and Test Control Notation 3* (*TTCN-3*) was released by the *European Telecommunications Standards Institute* (*ETSI*) in 2001 with major contributions by the Institute for Telematics, Lübeck, involving the author himself. Although TTCN-3 is considered the successor of TTCN-2 and although both languages share many basic concepts, they have totally different styles. In fact, TTCN-3 was redesigned from scratch. The main features of TTCN-3 are described in section 3.2.

The Inres Case Study. The concepts of TTCN-2 and TTCN-3 are illustrated by test suites for Inres, a service and protocol designed for educational purposes (Hogrefe, 1989). It is also used in chapter 5 for the description of the formal specification languages MSC and SDL.[1]

[1] Another introduction to TTCN-3, MSC, and SDL by the author with a consistent case study from process automation is published as Grabowski and Schmitt (2002); Grabowski et al. (2001, 2002).

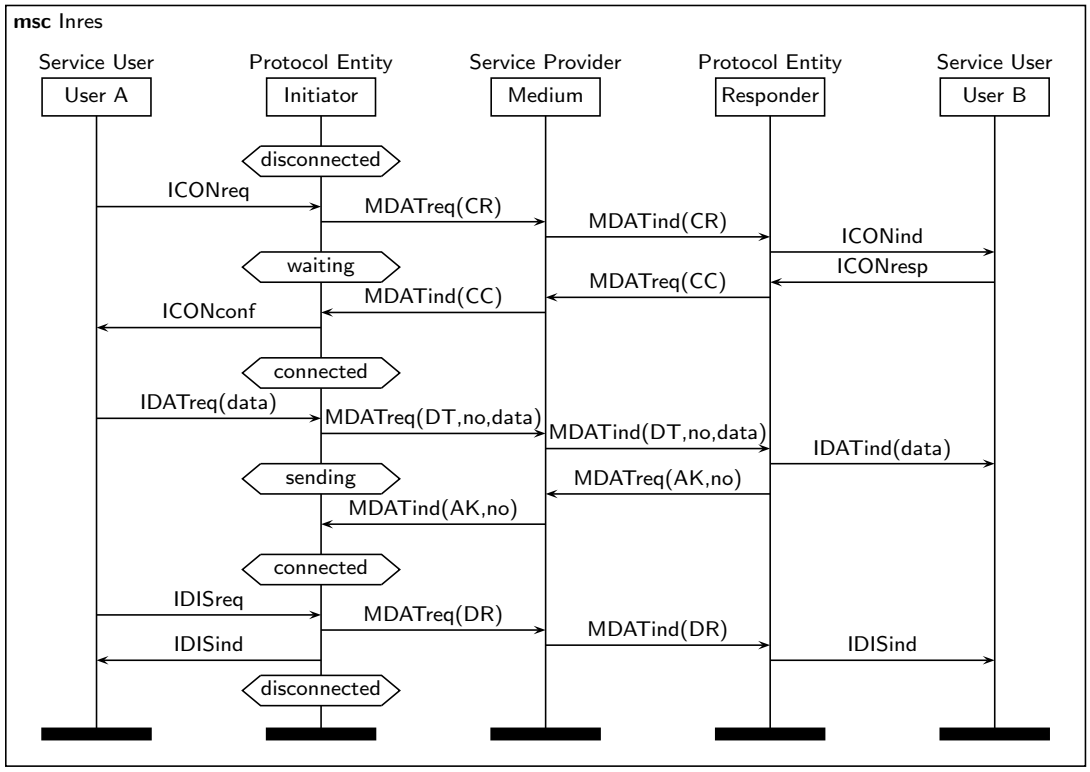


Figure 3.1: The Inres service and protocol

Inres – which stands for INitiator-RESponder – is a reliable, asymmetric and connection-oriented service on the OSI data link layer that ensures the safe transmission of data over an unreliable medium. For that purpose, a sequence number is transmitted along with each data. The responder protocol entity must acknowledge each data packet by the correct sequence number.

The Inres service comprises the three phases *connection establishment*, *data transfer*, and *connection release*. The message exchange that takes place when a service user A transmits one data packet to some service user B is shown in the *message sequence chart* (*MSC*) in figure 3.1.

The main features of TTCN-2 and TTCN-3 are illustrated by test suites for testing the conformance of *Initiator* protocol entity implementations. The local test method of the CTMF is chosen, i.e., both upper and lower tester reside inside the test system. The upper tester takes the role of *Service User A* and exchanges Inres ASPs with the SUT via Inres service access point *ISAP1*. The lower tester simulates the behavior of a *Responder* protocol entity and communicates with the SUT via service access point *MSAP2* of the *Medium* service provider. The conceptual architecture is shown in figure 3.2.

For simple comparison of both languages, the TTCN-2 and TTCN-3 conformance test suites for Inres are specified as syntactically and semantically equivalent as possible while taking into account the characteristics of both languages. In the following sections, only

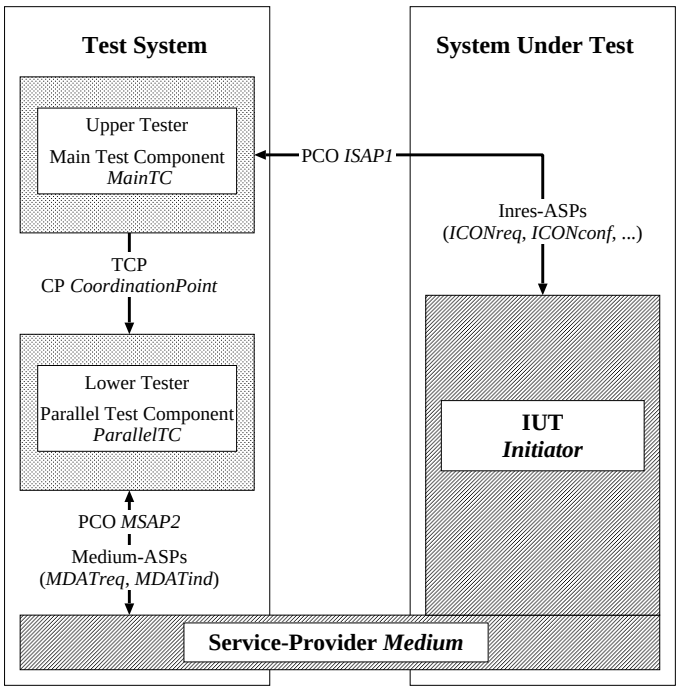


Figure 3.2: The local test method applied to Inres

simplified extracts are presented. Complete test suites can be found in appendices B and C.

3.1 The Tree and Tabular Combined Notation

The *Tree and Tabular Combined Notation*, second edition, (*TTCN-2*) is a language for the specification of abstract test suites for OSI protocol testing. It is published as the third part of *ISO/IEC International Standard 9646* (ISO/IEC, 1997).

A TTCN-2 document can have two semantically equivalent forms: The graphical representation, called *TTCN.GR*, is based on tables in which the test specifier fills in all test suite information. The machine processable form, called *TTCN.MP*, reflects the structures and contents of these tables in a text-based representation. TTCN.MP is hard to edit by hand and serves solely for the purpose of storing TTCN-2 test suites in a canonical way and for exchanging them between different tools. In the following, examples are given only in the graphical form.

A TTCN-2 test suite is structured into five parts:

- The *Test Suite Overview* part contains information on the purpose, extent, and structure of a test suite.
- The *Import Part* lists objects that are used in the test suite but defined in a different document.

Test Suite Structure			
Suite Name : TestsForInres			
Standards Ref :			
PICS Ref :			
PIXIT Ref :			
Test Method(s) : Local test method			
Comments :			
Test Group Reference	**Selection Ref**	**Test Group Objective**	**Page Nr**
BasicInterconnectionTests/		Determine whether there is sufficient conformance for interconnection to be possible	210
BehaviorTests/		Determine the extent to which dynamic conformance requirements are met	210
Detailed Comments :			

Test Case Index				
Test Group Reference	**Test Case Id**	**Selection Ref**	**Description**	**Page Nr**
BasicInterconnectionTests/	SingleDataTransfer			210
BehaviorTests/	DataLoss	InopportuneEvents		210
Detailed Comments :				

Figure 3.3: TTCN-2 – Test suite overview

- The *Declarations Part* contains definitions and declarations of all objects used in the other parts of the test suite.
- The *Constraints Part* defines the values that are to be sent and received by the tester.
- The *Dynamic Part* specifies the dynamic test behavior, i.e., the test outcomes.

For each part, TTCN-2 defines a set of table proformas. Depending on the type of table, a TTCN test suite can have one or more instances. In the following, the main concepts of each part are described and illustrated by simple examples.

3.1.1 Test Suite Overview and Import Part

In the *Test Suite Overview* part, general information on the purpose and content of a test suite are provided. This information includes the structure of the test suite, indexes of test cases and test steps, and exported objects that can be reused in other test suites.

In figure 3.3, two tables of the test suite overview part are shown. Their overall layout and their captions (printed in bold font) are prescribed by the TTCN-2 standard. The *Test Suite Structure* table specifies the name of the test suite, its supplementary documents (standards, protocol ICSs (PICSs), and protocol IXITs (PIXITs)), its test method, and its test groups. In the given example, two test groups, *BasicInterconnectionTests* and *BehaviorTests*, are defined. In the *Test Case Index* table, all test cases and their corresponding test groups are listed.

TTCN-2 allows to define selection expressions that apply to either a single test or a complete test group. A test case/test group is only executed if its selection expression

evaluates to true. This mechanism allows to choose test cases depending on the capabilities of the IUT. In figure 3.3, second table, test case *DataLoss* is tied to selection expression *InopportuneEvents* (its definition is given in a separate table in appendix B).

The *Import Part* consists of only one table which lists all objects (type definitions, test steps, etc.) that are imported from another source. Conceptionally, it is the counterpart of the *Test Suite Exports* table in the Test Suite Overview part in which all exportable objects are declared.

3.1.2 Declarations Part

The *Declarations part* comprises declarations and definitions for all objects used in the test suite, including data types, parameters, constants, variables, timers, auxiliary functions (operations), and elements of test component configurations.

Data Model. TTCN-2 comes along with various predefined simple types, including INTEGER, BOOLEAN, BIT/HEX/OCTETSTRING, and 12 types of character strings with varying character set. Based on these basic types, subtypes with restricted value range or complex data types can be constructed.

In accordance with OSI terminology, TTCN-2 distinguishes between data types, ASP types, and PDU types. Simple and structured data may appear inside ASPs and PDUs but they are not allowed to be used directly for send and receive test events. In addition, *coordination messages* (*CMs*) must be defined for communication between two test components within the tester. For each kind of type, TTCN-2 provides a distinct table proforma. Normally, each definition is made in a separate table but there are also compact proformas for multiple definitions to reduce the amount of tables.

In the telecommunication area, it has become common practice to define data types in the *Abstract Syntax Notation 1* (*ASN.1*; ITU-T, 1997a). Thus, TTCN-2 permits the use of ASN.1 as alternative to its own data language. Table proformas are provided to define ASN.1 data types inside the test suite or to refer to an external ASN.1 module.

In figure 3.4, two ASN.1 definitions are presented. In the first table, PDU type *InresPDU* is defined as a sequence with the three fields *iPDUType*, *seqNo*, and *iSDU* where the latter two are optional (depending on *iPDUType*; cf. figure 3.1). According to the test method shown in figure 3.2, a medium service provider is used for communication between the lower tester and the SUT. It exchanges ASPs of type *MDATreq* and *MDATind* with the lower tester in which *InresPDUs* are embedded. In the second table of figure 3.4, the definition of *MDATreq* is given.

By default, TTCN-2 does not make any assumptions on the ranges of integers and floats, since these details are tightly coupled with aspects of data encoding which is outside the scope of the language itself. However, the test specifier may assign *encoding rules* to the whole test suite or to single data types and PDU types. The semantics of these rules is determined in some external manner, e.g., by the test system. In figure 3.4, first table,

ASN.1 PDU Type Definition	
PDU Name	: InresPDU
PCO Type	:
Encoding Rule Name	: PER_BASIC_UNALIGNED_1997
Encoding Variation	:
Comments	: Apply Packed Encoding Rules
Type Definition	

SEQUENCE {
iPDUType InresPDUType, -- *CR*, *CC*, *DR*, *DT*, or *AK*
seqNo SequenceNumber OPTIONAL, -- *zero* or *one*
iSDU UserPDU OPTIONAL
}	

Detailed Comments : A User PDU on layer n+1 becomes an Inres SDU on layer n	

ASN.1 ASP Type Definition	
ASP Name	: MDATreq
PCO Type	: MediumSAP
Comments	:
Type Definition	
SEQUENCE { mSDU InresPDU }	
Detailed Comments : An Inres PDU on layer n becomes a Medium SDU on layer n-1	

Figure 3.4: TTCN-2 – ASN.1 type definitions

the specification of encoding rule *PER_BASIC_UNALIGNED_1997* means that PDUs of type *InresPDU* must be encoded according to the *ASN.1 Packed Encoding Rules*.

Parameters, Constants, and Variables. Test suites can be parameterized for re-use in different contexts. Thereby, properties of the IUT can be passed to a test suite and, e.g., used in test selection expressions. Test suite parameters are considered global constants. For each parameter, a reference to the corresponding entry in a PICS/PIXIT document must be given. Further global constants that are not derived from a PICS or PIXIT can be defined in a separate *Test Suite Constant Declaration* table.

Variables can be defined with two different scopes and life-times. *Test suite variables* exist during the execution of the whole test suite and thus can be used to pass information from one test case to another (e.g., test verdicts). Test suite variables are only accessible by the main test component. In contrast, the life-time of *test case variables* is restricted to each single test case. All test components, i.e., both the main and the parallel test components, obtain their own complete set of variable instances at the time of creation.

Test Component Configurations. TTCN-2 allows to define one or more *test component configurations* in a test suite. A test component configuration is characterized by a *main test component* (*MTC*), zero or more *parallel test components* (*PTCs*), and the communication links among these components and between the test components and the SUT. If no test component configuration is given, a default configuration with only one test component is assumed. Each test component executes in parallel. Test component configurations must be defined statically, i.e., the number of components and their

PCO Declarations			
PCO Name	**PCO Type**	**Role**	**Comments**
ISAP1	InitiatorSAP	UT	
MSAP2	MediumSAP	LT	
Detailed Comments :			

Coordination Point Declarations	
CP Name	**Comments**
CoordinationPoint	Message exchange between MTC and PTC
Detailed Comments :	

Test Component Declarations				
Component Name	**Component Role**	**Nr PCOs**	**Nr CPs**	**Comments**
MainTC	MTC	1	1	
ParallelTC	PTC	1	1	
Detailed Comments :				

Test Component Configuration Declaration			
Configuration Name : StandardConfiguration			
Comments :			
Components Used	**PCOs Used**	**CPs Used**	**Comments**
MainTC	ISAP1	CoordinationPoint	
ParallelTC	MSAP2	CoordinationPoint	
Detailed Comments :			

Figure 3.5: TTCN-2 – Test component configuration

communication links are fixed. However, all PTCs are created dynamically at execution time by the MTC (see section 3.1.4).

In TTCN-2, communication is based on asynchronous message exchange. A test component communicates with the SUT by one or more *points of control and observation* (*PCOs*). The semantic counterpart for communication between two test components is called *coordination point* (*CP*). Each PCO and CP is modeled by one input queue for incoming messages and one output queue for outgoing messages. Both queues have infinite capacity such that messages never get lost, even if they cannot be processed immediately.

In figure 3.5, a test component configuration is defined that corresponds to the local method presented in figure 3.2. The *PCO Declarations* table defines the two PCOs *ISAP1* and *MSAP2*. For each PCO, the corresponding role, i.e., *upper tester* (*UT*) or *lower tester* (*LT*), is specified. A CP, cleverly called *CoordinationPoint*, is specified in the *Coordination Point Declarations* table. The *Test Component Declarations* table defines the two components *MainTC* and *ParallelTC*, their roles (*MTC*/*PTC*) and their numbers of PCOs and CPs. Finally, the configuration is specified in the *Test Component Configuration Declaration* table that lists all test components involved and assigns concrete PCOs and CPs to them.[2]

[2]Obviously, the scattering and duplication of information in several tables causes a significant overhead and only pays off if more than one test component configuration is defined.

Test Suite Operations. TTCN-2 provides a number of predefined operators for arithmetic and boolean operations, comparisons, conversions (e.g., from HEXSTRING to INTEGER), for determining the presence of optional data fields, and for finding out the length/size of sequences and strings. If these operations are insufficient, the test specifier can define his own operations in terms of functions with input parameters and result value. Two different table types are provided for either a procedural or a textual (i.e., informal) description. For the first case, TTCN-2 provides a rudimentary imperative language that comprises assignments as well as `RETURN`, `IF`, `WHILE`, and `CASE` statements.

3.1.3 Constraints Part

In TTCN-2, test data, i.e., the concrete messages that are exchanged during test execution, are described by *constraints*. The constraints concept provides a simple way to organize and re-use test data. In analogy to the declarations part, TTCN-2 distinguishes between constraints for structured types, ASP types, PDU types, and CM types. For all kinds, similar table proformas are provided.

One of the strengths of TTCN-2 that makes it particularly suitable for test specification is the flexible way in which messages can be specified. In many cases, messages sent by the SUT are not exactly compared with a concrete value by the tester. For instance, some message parameters may not be relevant for a specific test purpose and thus do not have to be checked. More important, the responses of the SUT often depend on the test history or are even at random, e.g., when the SUT exchanges sequence numbers with the tester.

For that reason, constraints for receive events may make use of *matching mechanisms*. Instead of concrete values, special operators can be specified that match, e.g., any value, a list of single values, a range of values, the complement of a value list or all permutations of a given sequence. There are also operators for handling optional data and for imposing restrictions on the length of strings and sets.

In figure 3.6, three *ASN.1 Constraint Declaration* tables are shown. In the first table, a question mark is specified instead of a concrete value for data field *seqNo*. It means that any sequence number is accepted for a receive event that refers to constraint *DataTransfer*. Nevertheless, the value that is actually received can be retrieved during test execution. In figure 3.8, line number 4, the value for *seqNo* is stored in variable *seqNumber* and used for the successive reply (see line number 5, *Constraints Ref* column).

Structuring Concepts. Constraint descriptions can become very large and complex.[3] In order to reduce their size and improve readability, constraints can be structured in three ways:

- Constraint parameterization

[3] See chapter 9 for a detailed discussion on this topic.

ASN.1 ASP Constraint Declaration	
Constraint Name	: DataTransfer(data : UserPDU)
ASP Type	: MDATind
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU { iPDUType DT, seqNo ?, iSDU data } }	
Detailed Comments :	

ASN.1 PDU Constraint Declaration	
Constraint Name	: ConnectionConfirmation
PDU Type	: InresPDU
Derivation Path	:
Encoding Rule Name	:
Encoding Variation	:
Comments	: This constraint is used with constraint 'MediumDataRequest'
Constraint Value	
{ iPDUType CC }	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: MediumDataRequest(data : InresPDU)
ASP Type	: MDATreq
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU data }	
Detailed Comments :	

Figure 3.6: TTCN-2 – Constraints

- Constraint chaining
- Constraint derivation

In case of *constraint parameterization*, the parameter may either be a simple/structured data type or a PDU type. *Constraint chaining* means that a constraint refers to another constraint. Two types of chaining are distinguished: If there is an explicit (hard-coded) constraint reference in the *Constraint Value* definition, the constraints are chained *statically*. Constraint chaining is called *dynamic* if some value in a constraint is a formal parameter and a constraint reference is passed as actual parameter. *Constraint derivation* is useful if there are many similar constraints of a particular ASP, PDU, CM, or data type. In this case, one or more base constraints can be defined that specify a set of default values or wildcards for each field. Then, only those fields have to be specified in a modified constraint whose values deviate from the corresponding values in the base constraint.

Constraint parameterization is illustrated by constraint *DataTransfer* (figure 3.6, first table) that has *data* of type *UserPDU* as formal parameter. In figure 3.8, line 4, a reference to this constraint is made in column *Constraints Ref* with actual parameter *someUserPDU*. Dynamic constraint chaining is demonstrated by the second and third

Test Case Dynamic Behaviour					
Test Case Name : SingleDataTransfer					
Group : BasicInterconnectionTests/					
Purpose :					
Configuration : StandardConfiguration					
Default : MTCFailure					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		CREATE(ParallelTC:MediumAccess)			
2		START testCaseTimer			
3		+Preamble			
4		START supervisionTimer			restrict time of data transfer
5		ISAP1 ! IDATreq	InresDataRequest (someUserPDU)		data transfer
6		CoordinationPoint ? Notification	acknowledgmentSent	(PASS)	wait until 'ptc' has acknowledged the data
7		CANCEL supervisionTimer			cancel timer to avoid a timeout in the following
8		+Postamble			
9		? DONE(ParallelTC)		R	
10		ISAP1 ? IDISind	InresDisconnection-Indication	INCONC	
		Preamble			
11		ISAP1 ! ICONreq	InresConnectionRequest		
12		ISAP1 ? ICONconf	InresConnection-Confirmation		
13		ISAP1 ? IDISind	InresDisconnection-Indication	INCONC	
		Postamble			
14		ISAP1 ! IDISreq	InresDisconnection-Request		
15		ISAP1 ? IDISind	InresDisconnection-Indication		
Detailed Comments :					

Figure 3.7: TTCN-2 – Test case *SingleDataTransfer*

constraint in figure 3.6. ASP constraint *MediumDataRequest* expects an *InresPDU* as parameter. In test step *MediumAccess* (figure 3.8, line 3), constraint *ConnectionConfirmation* is passed as actual parameter.

3.1.4 Dynamic Part

In the dynamic part, the test outcomes are specified by means of test cases, test steps, and default behavior. A *test case* (see figure 3.7) defines the dynamic behavior of the MTC. *Test steps* either describe logically bounded sequences of test events or the behavior of PTCs (see figure 3.8). *Defaults* allow the handling of unexpected test events in an elegant way (see figure 3.9).

Control Flow. The dynamic behavior, i.e., the expected sequences of test events, is described in a tree-like structure. The temporal ordering of events is expressed by in-

Test Step Dynamic Behaviour					
Test Step Name : MediumAccess					
Group :					
Objective :					
Default : PTCFailure					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		MSAP2 ? MDATind	ConnectionRequest		
2		(receipt := 1)			first connection request
3		MSAP2 ! MDATreq	MediumDataRequest(ConnectionConfirmation)		
4	Loop	MSAP2 ? MDATind (seqNumber := MDATind.mSDU.seqNo)	DataTransfer(someUserPDU)	(PASS)	
5		MSAP2 ! MDATreq	DataAcknowledgment (seqNumber)		
6		CoordinationPoint ! Notification	acknowledgmentSent		inform the MTC that the data have been acknowledged
7		MSAP2 ? MDATind	DisconnectionRequest	PASS	
8		MSAP2 ? MDATind	ConnectionRequest		
9		[receipt <= maxRepetitions]			connection confirmation got lost
10		(receipt := receipt + 1)			resend it
11		MSAP2 ! MDATreq	MediumDataRequest ({ iPDUType CC })		
12		-> Loop			
13		[receipt > maxRepetitions]		FAIL	the initiator shall not resend its request that often
Detailed Comments :					

Figure 3.8: TTCN-2 – Test step *MediumAccess*

dentation of statements. Alternative test events are specified with the same amount of indentation.

In figure 3.7, the test events specified in lines 1 to 5 are executed/evaluated from top to bottom due to increasing indentation. The test events in line 6 and 10 are alternatives. If, at run-time, the first test events occurs, test execution proceeds in line 7; otherwise the test case terminates prematurely.

In order to describe loops, TTCN-2 provides a `REPEAT` statement. It executes a test step until a break condition is fulfilled. In addition, a `GOTO` statement (also specified as `->`) allows to jump to any point in the behavior description. The target of a `GOTO` statement is denoted by a label that is specified in the second column of the behavior table. In figure 3.8, the `GOTO` statement in line 12 makes the test component continue execution in line 4 (or in line 8 where an alternative event is specified).

Test steps can be embedded into a test case or another test step by means of an attachment statement (+*testStepName*). They are treated in a macro-like manner and can be parameterized by PCOs, ASPs, PDUs, or simple/structured data. Test steps can be defined either locally inside a test case description or by a separate *Test Step Dynamic*

Default Dynamic Behaviour					
Default Name : MTCFailure **Group** : Failures/ **Objective** : **Comments:** :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		ISAP1 ? OTHERWISE		FAIL	
2		? TIMEOUT		FAIL	
Detailed Comments :					

Figure 3.9: TTCN-2 – Default *MTCFailure*

Behaviour table.[4] In figure 3.7, two test steps, *preamble* and *postamble*, are used inside test case *SingleDataTransfer* (lines 3 and 8). They are defined locally in lines 11–13 and 14–15.

In a test case or test step description, the expected sequences of test events should be specified in a compact way. On the other hand, a tester should be able to react adequately to events which are unexpected or whose time of occurrence is not predictable. In order to avoid that all possible alternatives of test events have to be listed explicitly in a test case description, TTCN-2 provides a *default* mechanism.

In figure 3.9, default behavior *MTCFailure* is defined. It is used by test case *SingleDataTransfer* (figure 3.7) and makes test execution stop with verdict *fail* when a timeout occurs or some message is received at PCO *ISAP1* that is not handled explicitly.

TTCN-2 is based on a *snapshot semantics*. Whenever there are several alternative test events, the current state of the complete test system, i.e., the state of all PCOs, variables, etc., is considered as frozen. Then, all foreseen test events are checked from top to bottom. If none of the test events in question happened, a new snapshot of the system is taken and the alternative test events are checked again. This process is repeated until eventually one of the test events occurs.

Statements. Communication in TTCN-2 is based on the asynchronous exchange of messages via PCOs and CPs by means of send and receive operations. A sender continues processing immediately after executing a send operation, whereas a receive operation blocks a test component until the expected message eventually arrives.

The reception of a message by the tester is specified in the form *PCOorCP* `?` *Type* where *Type* is either an ASP, PDU, or CM type. The concrete message is specified by a constraint whose name is given in the *Constraints Ref* column in the dynamic behavior table. To deal with unforeseen test events, the keyword `OTHERWISE` might be used instead of a type. *PCOorCP* `?` `OTHERWISE` makes the tester accept any incoming event at the specified PCO or CP.

For send events, the notation *PCOorCP* `!` *Type* is used. Since the tester must behave deterministically, the associated constraint is not allowed to include special matching

[4] Throughout this thesis, the American spelling is used. The only exception is *behaviour* if it refers to a fixed term in a TTCN-2 table header.

symbols but only specific values. In the *remote test method* defined by ISO/IEC 9646, there is no way to control the IUT by an upper tester. Nevertheless, some action might be necessary to make the IUT issue a PDU or ASP. This action can be indicated by an *implicit send* event of the form `<IUT ! Type>` .

Test components can have local timers to control test execution. TTCN-2 provides various operations for setting and evaluating timers: The `START` and `CANCEL` operations activate and deactivate timers. The expiration of a timer is ascertained by a `TIMEOUT` statement. Finally, the `READTIMER` operation returns the time that has passed since the activation of the timer.

In figure 3.7, line 4, timer *supervisionTimer* is used to restrict the time of an Inres data transfer. If the timer expires before it is stopped in line 7, a timeout occurs. This event is handled in figure 3.9, line 2.

If there is more than one test component, the MTC is responsible for creating all parallel test components. For that purpose, `CREATE` statements must be specified at the beginning of the test case description. The MTC is only allowed to terminate after all PTCs have terminated. Otherwise, a test case error occurs. The MTC can check the status of the PTCs by `DONE` statements (see figure 3.7, line 9).

Test execution can depend on boolean expressions, called *qualifiers*. If a qualifier evaluates to true, execution continues with the subsequent statement; otherwise, the next alternative is chosen. In case all alternatives are qualifiers that evaluate to false, a test case error occurs.

In figure 3.8, lines 9 and 13, the number of connection confirmations that have been received is compared with the allowed number of repetitions. Depending on the result, test execution is continued (line 11) or stopped with the verdict `FAIL`.

Test Verdicts. A test verdict must be assigned to each test sequence of a test case or test component. Test verdicts are specified in the *Verdict* column of dynamic behavior tables. TTCN-2 allows to set three types of verdicts: `PASS`, `FAIL`, and `INCONC` (inconclusive). If a verdict is preliminary, its identifier is put into parentheses (e.g., `(PASS)`).

For each parallel test component, a local test verdict is maintained that is accessible as variable `R`. In the test case description for the MTC, `R` denotes a global test verdict. Setting a final verdict causes the immediate termination of the corresponding test component. If the MTC terminates, the complete test case execution is stopped. Based on the test verdicts of all test components, the final global test case verdict is computed. The necessary communication happens implicitly, i.e., it does not have to be specified in the test suite.

During test execution, it must be prevented that, e.g., a preliminary `(FAIL)` verdict is substituted by a final `PASS`, because once test execution fails, it fails forever. Thus, whenever a test verdict is to be set or updated, TTCN-2 applies a set of overwriting rules that take into account the current verdict.

3.2 The Testing and Test Control Notation

The *Testing and Test Control Notation version 3* (*TTCN-3*; see ETSI, 2002a) is a universal language for the specification and implementation of tests for distributed systems. Like its predecessors, TTCN-3 is used for functional black box testing. But in contrast to TTCN-2, it does not contain any language constructs specific to OSI conformance testing. TTCN-3 extends and generalizes many TTCN-2 concepts to open the language towards other testing methodologies and application domains such as the *Common Object Request Broker Architecture* (*CORBA*) or *Application Programming Interfaces* (*APIs*).

The TTCN-3 standard comprises a textual core language whose syntax resembles traditional imperative programming languages. In addition, ETSI European Standard 201 873 provides for the definition of *presentation formats*. Currently, there are two standardized formats: The *tabular presentation format* (*TFT*; ETSI, 2002b) is similar to TTCN-2 and helps TTCN users to migrate to the new version. The *graphical presentation format* (*GFT*; ETSI, 2002c) is based on a dialect of *Message Sequence Chart* (*MSC*; see section 5.1) with testing-specific extensions.

3.2.1 Modules and Groups

The topmost structuring concept in TTCN-3 is the *module*. A module may define a complete abstract test suite or a library that can be used by another module. By analogy with TTCN-2 test suites, modules can be parameterized to re-use them in different contexts.

Modules consist of two parts: The *definitions part* includes definitions of data types, constants, communication data (messages, procedure signatures, and templates), test configuration elements (ports, components), and the dynamic behavior (test cases, functions, and altsteps). The *control part* is the main program of a TTCN-3 module. It allows to state explicitly the order in which test cases are to be executed. Moreover, execution of single test cases can be made dependent on some selection criteria, e.g., the outcome of a preceding test case.

A module can import arbitrary definitions from other modules. There is no explicit export construct in TTCN-3; all definitions in the module definitions part can be imported from another module.

Within a module, definitions can be arranged in nested *groups* in order to enhance readability and structure the test suite with regard to logical aspects. Groups do not define scopes in general, i.e., it is not allowed to define a constant twice in distinct groups. However, it is possible to refer to a complete group when importing definitions from another module.

In figure 3.10, a TTCN-3 module for testing conformance of an Inres Initiator protocol entity is presented. Its definition part starts with an `import` statement (line 2–5) to adopt data type *UserPDU* and constant *someUserPDU* from an external module called

```
 1: module TestsForInres( integer maxRepetitions, boolean testInopportuneEvents ) {
 2:    import from ServiceUser language "ASN.1:1997" {
 3:       type UserPDU;
 4:       const someUserPDU;
 5:    }
 6:    group BasicDefinitions {
 7:       type enumerated InresPDUType { CR(1), CC(2), DR(3), DT(4), AK(5) };
 8:       type enumerated SequenceNumber { zero(0), one(1) };
 9:       type record InresPDU {
10:          InresPDUType iPDUType,
11:          SequenceNumber seqNo optional,
12:          UserPDU iSDU optional
13:       }
14:    } with { encode "PER-BASIC-UNALIGNED:1997" }
15:    const float maxTestCaseTime := 50;
16:    ... further definitions ...
17:    control {
18:       var verdicttype   overallVerdict := pass;
19:       overallVerdict := execute( SingleDataTransfer(), maxTestCaseTime );
20:       if ( overallVerdict == pass and testInopportuneEvents == true ) {
21:          overallVerdict := execute( DataLoss() );
22:       }
23:    }
24: } with { encode "BER:1997" }
```

Figure 3.10: TTCN-3 – Module *TestsForInres*

ServiceUser. All data type definitions that are needed to describe an Inres PDU are combined in group *BasicDefinitions* (lines 6–14). Thereafter, a global constant (*maxTestCaseTime*) is declared in line 15. Module *TestsForInres* includes many more definitions. For better readability and comprehension, these definitions are presented separately in figures 3.11–3.16.

In the module control part (lines 17–23), test case *SingleDataTransfer* is executed first. Depending on whether its execution has been successful (test verdict is *pass*) and module parameter *testInopportuneEvents* equals true, a second test case (*DataLoss*) is invoked.

3.2.2 Data Model

TTCN-3 defines its own data model. It covers many basic types known from programming languages like *integer*, *char*, *universal char* (as defined in ISO/IEC 10646, 1993), *float*, *boolean*, and various types of strings that differ in the character set allowed for their elements. Furthermore, a special type handling test verdicts (*verdicttype*) is provided. TTCN-3 does not support dynamic data structures and thus does not have a pointer concept. However, variables of the type *address* can be used for references to entities inside the SUT.

The test specifier can define different structured types based on *enumerations*, *records*, *sets, arrays,* and *unions*. Both records and sets allow the definition of *optional* fields.

Records and sets are equal except that the ordering of fields is not significant in sets, an aspect which is relevant for data encoding. For the same reason, records and sets with all fields being of the same type are treated separately. Semantically, they are equal to ordered resp. unordered arrays. In situations where data have to be handled whose exact type is unknown (e.g., PDUs), they can be assigned to a variable of type *anytype*, a short-hand for a union of all known types in a TTCN-3 module. Moreover, data structures are allowed to be defined recursively – a compensation for the lack of dynamic data structures.

TTCN-3 comes along with a large set of predefined functions that allow to convert a data value into a value of another type. There are also functions for retrieving the number of elements in a string, record, or set, for checking the presence of optional fields in a record, and for generating random numbers.

Subtypes of both basic and structured types can be defined by imposing restrictions on the set of valid values. For that purpose, the user may specify value ranges, values lists or length restrictions. As with its predecessor, data encoding is outside the scope of the TTCN-3. However, the test specifier may assign optional *encoding attributes* to modules, groups, type definitions, or single fields in record/set types.

In figure 3.10, two encoding rules are specified. The encoding attribute in line 24 states that the *ASN.1 Basic Encoding Rules* are applied to all data used inside module *Tests-ForInres* by default. Only Inres PDUs and their fields are transmitted based on *ASN.1 Packed Encoding Rules* (line 14).

Even though the TTCN-3 data concept suits most applications, it might be of advantage or necessary to use data descriptions provided in the implementation or specification language of the SUT. When importing definitions from an external module, a module language different from TTCN-3 can be specified. For telecommunication applications, existing ASN.1 data descriptions can be reused. For testing of distributed systems that are based on the CORBA middleware platform, a mapping of IDL interface specifications to TTCN-3 is defined in Ebner et al. (2002).

In figure 3.10, *ServiceUser* is declared to be an ASN.1 module. According to the ASN.1 transformation rules given in the TTCN-3 standard, *UserPDU* and *someUserPDU* are transformed into an equivalent TTCN-3 type and constant respectively.

3.2.3 Communication

In TTCN-3, communication within the test system and between the tester and the SUT can be either message-based or procedure-based.

Messages in TTCN-3 correspond to ASPs, PDUs, and coordination messages in TTCN-2. They are exchanged asynchronously by `send` and `receive` operations. A sender immediately continues processing after executing a `send` operation, whereas a `receive` operation blocks a component until the expected message eventually arrives. Messages are specified just as common data types (typically as records).

Procedure-based communication is required for testing, e.g., CORBA or DCOM (Distributed Common Object Model) platforms. A remote procedure is invoked by a `call` operation. Incoming calls are awaited at the callee side by a blocking `getcall` operation. Conceptionally, procedure calls can be considered both non-blocking and blocking with regard to the caller. In the latter case, the callee is expected to conclude procedure execution with a `reply` operation. The caller handles the answer with a `getreply` operation that is directly following its `call` operation.

Procedures can have an arbitrary number of parameters (with call-by-value and call-by-reference semantics) as well as a dedicated return value by which information can be exchanged between caller and callee. Moreover, procedures can be declared to raise exceptions that a caller should be able to catch. In a TTCN-3 test suite, the signature of a procedure, i.e., its interface definition, is required to check the semantics of corresponding communication operations.

In TTCN-3, communication takes place over connections that are terminated by communication endpoints. These endpoints are called *ports*. Any communication operation refers to a port rather than to the connection itself. Port types can be defined for message-based, procedure-based or mixed communication. In contrast to PCOs in TTCN-2, ports are directional. A port type is characterized by the set of valid messages and/or procedure types together with the direction (in/out/inout) for each individual message/procedure.

A port is modeled by a queue with infinite capacity to handle incoming messages/procedure calls, even if they are not processed directly by the tester. The above-mentioned `receive`, `getcall`, `getreply`, and `catch` operations check only the first element in the queue. However, a `trigger` operation can be used instead of `receive` that consumes all messages from the input queue until eventually a message with a certain property is found. Port queues can be cleared during execution. In addition, communication can be suppressed at a specific port and resumed later.

In figure 3.11(a), definitions for message-based communication are presented. *ICONreq* and *IDATreq* (lines 1 and 2) are – among others – two messages (ASPs) that can be sent to the Initiator entity of the Inres protocol. A port type definition is given in lines 3–6. The keywords `out` and `in` indicate that messages *ICONreq*, *IDATreq*, and *IDISreq* can be sent and *ICONconf* and *IDISind* can be received by a corresponding port instance. For communication with the Medium, similar definitions are made in lines 7–12. A concrete message exchange is described in the test case shown in figure 3.14. In lines 11, 12, 14, 19, and 20 various messages are sent and received at port *ISAP1* which is of type *InitiatorSAP*.

A simple example of procedure-based communication is introduced in figure 3.11(b). In line 1, procedure *acknowledgmentSent* is defined which has neither a parameter nor a return value. It is used for coordination between the components of the tester. Conceptionally, it resides at the MTC.[5] Two contrary port types are defined for it: *PortAtMTC*

[5] Please note that the procedure is not implemented as such. Instead, only its invocation and termination is modeled by the MTC by `getcall` and `reply` operations. The procedure-based communication in the given example is only made for illustration purposes.

```
1:  type record ICONreq {};
2:  type record IDATreq { UserPDU iSDU };
3:  type port InitiatorSAP message {
4:      out ICONreq, IDATreq, IDISreq;
5:      in ICONconf, IDISind;
6:  }
7:  type record MDATreq { InresPDU mSDU };
8:  type record MDATind { InresPDU mSDU };
9:  type port MediumSAP message {
10:     in MDATind;
11:     out MDATreq;
12: }
```

(a) Message-based

```
1: signature acknowledgmentSent();
2: type port PortAtMTC procedure {
3:     in acknowledgmentSent;
4: }
5: type port PortAtPTC procedure {
6:     out acknowledgmentSent;
7: }
```

(b) Procedure-based

Figure 3.11: TTCN-3 – Communication

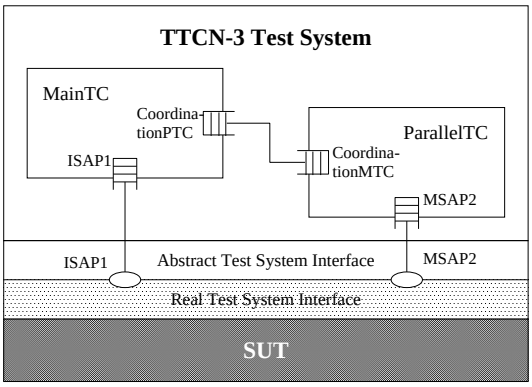
(lines 2–4) accepts incoming calls, whereas *PortAtPTC* (lines 5–7) is used to invoke the remote procedure. A concrete procedure call is realized in figure 3.14, lines 15 and 16, and figure 3.15, lines 21 and 22.

3.2.4 Test Configurations

TTCN-3 supports the specification of distributed test architectures. A test configuration is made up of a set of *test components* along with their *ports*, an abstract *test system interface*, connections between the ports of the test components, and associations between the ports and the test system interface. Unlike TTCN-2, TTCN-3 does not require test configurations to be declared statically but allows to modify them dynamically at run-time. This means, ports can be connected with and disconnected from other ports at any time during test execution. Moreover, ports can be connected in a one-to-many relationship to allow multicasts.

In figure 3.12(a), a conceptual view of the test configuration used in the Inres example is given. The tester consists of two test components called *MainTC* and *ParallelTC*. They communicate with each other via a connection established between the ports *CoordinationPTC* and *CoordinationMTC*. In addition, one port in each test component, namely *ISAP1* and *MSAP2*, is mapped to a port with the same name of the abstract test system interface. TTCN-3 abstracts from implementation issues such as encoding. Therefore, conceptionally the ports of the tester are not directly linked with the SUT itself. Instead, it is assumed that interaction with the SUT is realized by a *real test system interface* which is outside the scope of TTCN-3.

Figure 3.12(b) presents TTCN-3 test component definitions for *MainTC*, *ParallelTC* and the abstract test system (*TestSystem*) which is defined just like a common test component type. In addition to an arbitrary number of ports, a TTCN-3 test component can have its own set of local timers, variables and constants. For example, any component of type *MainTC* has a *supervisionTimer* at its disposal (line 4).



(a) Conceptual view

```
1:  type component MainTC {
2:      port InitiatorSAP ISAP1;
3:      port PortAtMTC CoordinationPTC;
4:      timer supervisionTimer;
5:  }
6:  type component ParallelTC {
7:      port MediumSAP MSAP2;
8:      port PortAtPTC CoordinationMTC;
9:  }
10: type component TestSystem {
11:     port InitiatorSAP ISAP1;
12:     port MediumSAP MSAP2;
13: }
```

(b) Component type definitions

Figure 3.12: TTCN-3 – Test configuration

Each test configuration consists of exactly one *main test component* (MTC) that is created implicitly when a test case is started. It behaves according to the statements in the test case description. One or more additional *parallel test components* (PTCs) can be created explicitly during test execution — either by the MTC or another PTC. When a PTC is started, it is associated with a *function* that describes its behavior.

All PTCs run independently from each other, i.e., termination of a test component does not cause its children to halt as well. A test component stops either implicitly when leaving its associated function or explicitly by executing a `stop` statement. It may also be stopped remotely by another test component. Test case execution comes to a complete halt if the MTC terminates. TTCN-3 provides two operations for checking the status of a test component: The `running` operation can be used to test whether a particular component (or all components) has completed execution, whereas the blocking `done` operation makes its caller wait until a designated test component eventually stops.

In the module control part of the TTCN-3 example test suite (figure 3.10, line 19), test case *SingleDataTransfer* is executed. Its definition is given in figure 3.14. According to its signature (line 1), it runs on a component of type *MainTC* and conforms to the abstract interface *TestSystem*. In line 4, a parallel test component of type *ParallelTC* is created. Thereafter, the ports *ISAP1* and *MSAP2* of the MTC (denoted by component reference *self*) and the PTC (denoted by *ptc*) are mapped to the ports of the test system interface (lines 5–6) and a connection is established between the MTC and the PTC (line 7). Execution of the PTC is actually started in line 8 by assigning function *MediumAccess* to it.

Before test case *SingleDataTransfer* ends, the MTC makes sure that function *MediumAccess* running on *ptc* has already terminated. This check is necessary to ensure that all communication between the PTC and the SUT has taken place. Execution of the MTC is blocked in line 21 until all PTCs (only one PTC in this example) have completed.

```
 1: template MDATind ConnectionRequest := {
 2:     mSDU := { iPDUType := CR, seqNo := omit, iSDU := omit }
 3: }
 4: template InresPDU ConnectionConfirmation := {   // this template is used with
 5:     iPDUType := CC, seqNo := omit, iSDU := omit   // template 'MediumDataRequest'
 6: }
 7: template MDATreq MediumDataRequest( template InresPDU data ) := {
 8:     mSDU := data
 9: }
10: template MDATind DataTransfer( UserPDU data ) := {
11:     mSDU := { iPDUType := DT, seqNo := ?, iSDU := data }
12: }
```

Figure 3.13: TTCN-3 – Templates

3.2.5 Templates

The description of messages and procedures is made by *templates*. The template concept is an extension of the *constraint* concept in TTCN-2. Templates can be defined for both message- and procedure-based communication. For receiving messages and incoming procedure calls/replies, TTCN-3 provides more or less the same *matching mechanisms* as TTCN-2. However, the set of operators for string matching has been extended to obtain the expressiveness of regular expressions.

Templates can be structured in the same way as constraints in TTCN-2, i.e., by parameterization, referencing, and modification (derivation). But while TTCN-2 distinguishes between a dynamic part and a constraint part, TTCN-3 does not enforce a clear separation between the control and data aspects of a test case — templates can either be referenced or specified inline within a test case or function. The latter alternative is suitable for messages and procedures with no or only a few fields/parameters where a standalone template definition means unnecessary expense and aggravates readability.

In figure 3.13, various templates are defined that are used in function *MediumAccess* (figure 3.15). Template *ConnectionRequest* (lines 1–3) specifies a message of type *MDATind* where the fields *mSDU.seqNO* and *mSDU.iSDU* shall have no value. It is used in combination with a `receive` statement in figure 3.15, line 5.

The two templates *ConnectionConfirmation* and *MediumDataRequest* (figure 3.13, lines 4–9) illustrate the dynamic chaining of templates. *MediumDataRequest* is parameterized by a template of type *InresPDU*. In figure 3.15, line 7, it is instantiated with template *ConnectionConfirmation* as actual parameter. Template *DataTransfer* (line 10–12) makes use of a simple matching mechanism. Operator "?" states that any value for *seqNo* is acceptable in an incoming message.

Many templates are defined inline in test case *SingleDataTransfer* (figure 3.14, lines 11, 12, 15, 16, 19, and 20) and function *MediumAccess* (figure 3.15, lines 11, 21, 22, 23). In particular, many messages exchanged with the SUT via port *ISAP1* are distinguished by their type only. Hence, there is no need for template definitions whose bodies would only consist of empty brackets syntactically.

```
 1: testcase SingleDataTransfer() runs on MainTC system TestSystem {
 2:     var ParallelTC   ptc;
 3:     var default   def1, def2;
 4:     ptc := ParallelTC.create;
 5:     map( self:ISAP1, system:ISAP1 );
 6:     map( ptc:MSAP2, system:MSAP2 );
 7:     connect( self:CoordinationPTC, ptc:CoordinationMTC );
 8:     ptc.start( MediumAccess() );
 9:     def1 := activate( MTCFailure() );
10:     def2 := activate( ReceptionIDISind( inconc ) );
11:     ISAP1.send( ICONreq : {} );   // connection request
12:     ISAP1.receive( ICONconf : {} );   // connection confirmation
13:     supervisionTimer.start( maxTransferTime );   // restrict time of data transfer
14:     ISAP1.send( InresDataRequest( someUserPDU ) );   // data transfer
15:     CoordinationPTC.getcall( acknowledgmentSent : {} );
16:     CoordinationPTC.reply( acknowledgmentSent : {} );
17:     supervisionTimer.stop;   // cancel timer to avoid a timeout in the following
18:     deactivate( def2 );
19:     ISAP1.send( IDISreq : {} );   // disconnection request
20:     ISAP1.receive( IDISind : {} );   // disconnection indication
21:     all component.done;
22:     setverdict( pass );
23: }
```

Figure 3.14: TTCN-3 – Test case *SingleDataTransfer*

3.2.6 Behavior Descriptions

In TTCN-3, the functional behavior is described by test cases, functions, and altsteps. Like in TTCN-2, a *test case* describes the dynamic behavior of an MTC (see test case *SingleDataTransfer* in figure 3.14). *Functions* in TTCN-3 correspond to TTCN-2 test steps and test suite operations. In figure 3.15, function *MediumAccess* is shown that runs on a test component of type *ParallelTC*. The *altstep* mechanism is similar to TTCN-2 defaults.

Control Structures. TTCN-3 supports most control structures known from imperative programming languages. These are `if ... else`, `for`, `while`, and `do ... while`. For a simpler transformation of existing TTCN-2 test suites, `goto` statements can be used to jump to a labeled position in the program code.

In TTCN-2, test cases are specified in a tree-like notation. If different branches contain a common test sequence (e.g., a preamble), its statements have to be duplicated or put in a separate test step. In contrast, behavior descriptions in TTCN-3 are described in a sequential manner. Alternative behavior is described by an `alt` statement. Each alternative within an `alt` statement consists of (a) an optional boolean expression (b) a guard operation which may by either a `done` operation, a `timeout` operation, or any receiving operation, and (c) a statement block. The latter is executed if the expression evaluates to true (or no expression is specified) and the guard operation can be executed.

```
 1: function MediumAccess() runs on ParallelTC {
 2:     var integer   receipt;
 3:     var default   def := activate( PTCFailure() );
 4:     var MDATind   indication;
 5:     MSAP2.receive( ConnectionRequest );
 6:     receipt := 1;   // first (received) connection request of the initiator
 7:     MSAP2.send( MediumDataRequest( ConnectionConfirmation ) );
 8:     alt {
 9:       [ receipt <= maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
10:           receipt := receipt + 1;
11:           MSAP2.send( MediumDataRequest( { CC, omit, omit } ) );
12:           repeat;
13:         }
14:       [ receipt > maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
15:           setverdict( fail );
16:           stop;
17:         }
18:       [ ] MSAP2.receive( DataTransfer( someUserPDU ) ) -> value indication { /*empty*/ }
19:     }
20:     MSAP2.send( DataAcknowledgment( indication.mSDU.seqNo ) );
21:     CoordinationMTC.call( acknowledgmentSent : {} );
22:     CoordinationMTC.getreply( acknowledgmentSent : {} );
23:     MSAP2.receive( MDATind : { mSDU := { DR, omit, omit } } );
24:     setverdict( pass );   // disconnection request
25: }
```

Figure 3.15: TTCN-3 – Function *MediumAccess*

Similarly to TTCN-2, TTCN-3 defines a snapshot semantics for the processing of `alt` statements, i.e., the state of a test component is recorded before the alternatives are evaluated from top to bottom. If none of the alternatives can be executed, another snapshot is taken.

In figure 3.15, line 8–19, an `alt` statement with three alternatives is specified. It handles different messages sent by the SUT in response to a preceding connection confirmation (line 7). The first two alternatives consider the case that the confirmation got lost and hence the SUT resends its connection request. As long as the number of requests is less than or equal to constant *maxRepetitions* (line 9), the test component confirms the request once again.

By using the `repeat` statement in line 12, a re-evaluation of the whole `alt` statement is caused. However, if *receipt* > *maxRepetitions* (line 14), the SUT is not allowed to send another connection request and hence the test case fails. In the normal case, the tester receives a data transfer message (line 18) and test case execution is continued after the `alt` statements.

If a test component controls more than one port, the exact order in which messages and procedure calls arrive might be unpredictable, for instance, if the SUT broadcasts a message. Instead of listing all possible sequences in terms of a large `alt` statement, an `interleave` statement can be used. Syntactically, an `interleave` statement is similar to an `alt` statement but it is not allowed to guard a branch by a boolean expression.

Timers. Test components can have local timers to control test execution. TTCN-3 provides various operations for setting and evaluating timers: The `start` and `stop` operations activate and deactivate timers. The expiration of a timer is ascertained by a `timeout` statement. The current status of a timer can be retrieved by a `running` operation. In contrast to `timeout` which blocks the execution of a test component until the timeout occurs eventually, the `running` operation returns a boolean value instantaneously which can be evaluated in a condition. Finally, the `read` operation returns the time that has passed since the activation of the timer.

In figure 3.14, line 13, timer *supervisionTimer* is used to restrict the time of a data transfer of the SUT. If the timer expires before it is stopped in line 17, a timeout occurs which is handled in figure 3.16, line 6 (see the one but next paragraph for a description of the *altstep* concept). The execution of a complete test case can also be limited by an implicit timer. In figure 3.10, line 19, test case *SingleDataTransfer* is invoked. If its execution time exceeds *maxTestCaseTime*, all test components are stopped and the test case fails.

Test Verdicts. Test verdicts are objects of type *verdicttype*. They can take one out of five different values: `pass`, `fail`, `inconc`, `none`, and `error`. Each test component maintains its own implicit local verdict. The value of this local verdict can be set and retrieved by `setverdict` and `getverdict` operations where `setverdict` applies predefined overwriting rules to ensure that, e.g., a `fail` test verdict does not become a `pass` during test case execution. In addition, there is a global verdict that is updated implicitly according to the overwriting rules whenever a test component terminates. If a test case ends, the global verdict is returned. In contrast to TTCN-2, setting a verdict does not stop the execution of a test component. Moreover, there are no preliminary test verdicts in TTCN-3.

Altsteps. In TTCN-3, the default mechanism of TTCN-2 has been replaced and extended by the *altstep* concept. Altsteps are a collection of alternatives that are taken into account whenever a test component awaits a response from the SUT or another test component or the expiration of a timer. Their syntax is identical to alternatives within an `alt` statement (see section 3.2.6).

In figure 3.16, altstep *MTCFailure* is defined. Like its TTCN-2 counterpart, it makes test execution fail if a message is received at port *ISAP1* that is not handled elsewhere or a timer expires. Altstep *ReceptionIDISind* (same figure) illustrates the definition of a parameterized altstep. Depending on variable *result*, the reception of message *IDISind* leads to different test verdicts.

Altsteps can be activated and deactivated at any time during test execution. They can also be invoked explicitly from within a single `alt` statement. In figure 3.14, lines 9 and 10, the two above-mentioned altsteps are considered for the successive execution. In line 18, *ReceptionIDISind* is deactivated again because a disconnection indication is not an undesirable event any longer. Since a parameterized altstep can be instantiated

```
 1: altstep MTCFailure() runs on MainTC {
 2:     [ ] ISAP1.receive {
 3:           setverdict( fail );
 4:           stop;
 5:         }
 6:     [ ] any timer.timeout {
 7:           setverdict( fail );
 8:           stop;
 9:         }
10: }
11: altstep ReceptionIDISind( verdicttype result ) runs on MainTC {
12:     [ ] ISAP1.receive( IDISind : {} ) {
13:           setverdict( result );
14:           stop;
15:         }
16: }
```

Figure 3.16: TTCN-3 – Altsteps *MTCFailure* and *ReceptionIDISind*

several times with different values, the `activate` statement returns a handle of type *default*. This handle must be specified in a `deactivate` statement.

3.2.7 Development Tools

The first version of TTCN-3 was released as a European standard in July 2001. A revised edition was developed in 2002 by ETSI. Further extensions and corrections are going to be published in the future depending on user requests.

Though the standard is still rather new, there are already a lot of tools available for the TTCN-3 core language and the standardized presentation formats (Testing Technologies, 2002; Da Vinci Communications, 2002; Telelogic, 2002b). These tools facilitate the editing, compilation, debugging, and execution of TTCN-3 modules.

In parallel to the development of the TTCN-3 standard, a syntax checker has been developed by the author. The implementation and application of the parser made it possible to detect ambiguities among different language constructs, errors in the EBNF grammar, and inconsistencies between the language description and corresponding examples in the standard document. It also helped to address semantic issues as many of TTCN-3's static semantic rules are hard-coded in the grammar. The work on the TTCN-3 syntax checker has proven invaluable for TTCN-3 itself as it allowed to uncover problems at a very early stage of standardization.

3.3 Discussion

TTCN-3 symbolizes a major step towards a universal test language. It supports many different platforms and application areas by providing both message- and procedure-based communication, dynamic concurrent test configurations, and interfaces for exter-

nal data languages and encoding rules. In addition, TTCN-3 breaks with some terminology of TTCN-2 which focuses too much on OSI conformance testing.

Despite the benefits of TTCN-3, there are still various language elements and concepts that could be improved or generalized. A few possible enhancements are sketched in the following.

Object-oriented Language Model. TTCN-3 provides several special data types, e.g., component, port, and verdict, with a set of predefined operations. Though an object-oriented notation (*object.method*) is used for these operations, the language description of TTCN-3 does not reflect this view. An object-oriented model might be helpful to describe the language elements and their relationships.

Clarification of the Role of Templates, Data Types, and Data Values. The semantic classification of templates in TTCN-3 is rather vague. If a template of type T does not make use of any matching mechanisms, it can be considered an expression of type T. However, if a matching operator is used, the template can be considered a subtype T' of T. The main characteristic of this subtype T' is that it can only be decided at run-time whether a data value of type T is also in the domain of type T'. Obviously, such a check consumes a lot of computation time and should be avoided if possible.

With regard to data types, the situation is similar: For most data types, type checks can be made statically at compile-time. However, TTCN-3 allows the definition of subtypes T' (based on type T) with length or value range restrictions. Additional run-time checks are required whenever the value of a variable of type T is assigned to a variable of type T'.

A rigorous approach to clarify the role of templates would be to replace the existing template concept by a new class of data types and map the current concepts as follows:

Existing concepts		**New concepts**
data type	⇒	data type
subtype	⇒	dynamic data type (with run-time checks)
template w/ matching mechanisms	⇒	dynamic data type (with run-time checks)
template w/o matching mechanisms	⇒	data expression
data expression	⇒	data expression

Enhanced Matching Mechanisms. TTCN-3 allows characters patterns in templates to define the format of characters strings. These character patterns have the same expressive power as regular expressions. On the level of structured types, less powerful matching mechanisms are provided. In templates for arrays or sets/records of a single type, the matching operator "`*`" can be used as a wildcard for a sequence of zero or more elements (Example: `{ 1, *, 3 }`) .[6] However, there is no way to express, for instance, that a record with variable length shall consist of only one particular element. Listing all elements explicitly is not possible due to the unknown size of the record.

Hence, regular expressions should also be available for arrays, sets, and records. For example, the following notation might be used inside templates: *x*`#(`*min*,*max*`)` matches

[6] Please note that TTCN-3 has two different interpretations for the "*" symbol: When used on the highest level inside an array, its meaning is *AnyElementsOrNone*, otherwise it means *AnyValueOrNone*.

with at least *min* and at most *max* occurrences of x where x is either a single element or a group of sequential elements. If *min* or *max* is left unspecified, 0 or ∞ is used as default. Sequences of elements can be grouped by `< ... >`. For the description of alternatives, the existing notation for value lists is adopted.

```
var integer myInt := 8;
template record of integer RegExp :=
   { <1, 2>#(2,5), 3#(,), 4, (5, 6, 7), myInt, ? };
```

The example above illustrates what regular expressions for arrays, records, and sets could look like in TTCN-3. Template *RegExp* would match a record of integers which consists of the following elements: at least two and at most five times 1 followed by 2; zero or more 3's; a single 4; either 5, 6 or 7; the value of variable *myInt*; an arbitrary number.

Time Constraints. Timers allow to control the temporal execution of test events. Typically, they are used to postpone the execution of a send event or to check that a receive event took place in time. If a receive event shall happen within a given period of time (with lower and upper boundary), the TTCN-3 behavior description becomes rather complex: First, a timer is started that must expire before the receive event happens. As soon as the timer expires, it is re-started. Then, the receive event must occur before the timer expires. In total, two `alt` statements are required.

In situations like these, timer operations as an explicit means to supervise execution are not appropriate. Instead, a declarative approach where single operations or whole statement blocks are annotated with time constraints is preferable. Time constraints have already been introduced into the latest revision of Message Sequence Chart (section 5.1). A real-time extension for TTCN-3, called *Timed TTCN-3*, is proposed by Dai et al. (2002).

External Clocks. TTCN-3 has no notion of external clocks that are available on the test system. This makes it impossible to specify a test module where the execution of test cases is triggered by a specific time or date. Therefore, if a test engineer wants to execute his tests at 2 AM (because test execution within a production environment shall not affect the work of other users), then this constraint must be handled directly by the test system as it is outside the scope of TTCN-3.

On the other hand, external clocks could be integrated easily in TTCN-3 by a `now` operator that returns the current time and date. Since the test system may consist of several hardware devices, each test component as well as the module control part should have its own local clock, i.e., `now` may return totally different results even when executed in parallel on different test components. A solution for external clocks has also been integrated into Timed TTCN-3.

Nested Modules as a Replacement of the Group Concept. Modules are the top-level structuring element in TTCN-3. They serve two purposes: combining related data definitions, test configurations, communications data, and behavior descriptions in a logical unit; and controlling the execution of test cases by means of the module control part. Unfortunately, modules cannot be nested. Definitions can be combined in

```
1:  module TestsForInres( integer maxNoOfProcesses, boolean testInopportuneEvents ) {
2:      ...
3:      control {
4:          var verdicttype   v := pass, v1 := pass, v2 := pass;
5:          par ( maxNoOfTestCases ) {
6:              basic [ true ] {
7:                  v := execute( BasicInterconnectionTest() );
8:              }
9:              cap1 [ v == pass and basic == true ] {
10:                 v1 := execute( CapabilityTest1() );
11:             }
12:             cap2 [ v == pass and basic == true and testInopportuneEvents == true ] {
13:                 v2 := execute( CapabilityTest2() );
14:             }
15:         }
16:     }
17: }
```

Figure 3.17: Feature proposal – The *par* statement

groups but a group does not define a new scope and has no semantics except when being imported by another module. Moreover, TTCN-3 assumes that the entire test execution is controlled by the control part of the current module and some auxiliary functions.

Parallel Execution of Test Cases. The selection of test cases and the order in which they are executed is specified in the module control part. Unfortunately, the execution of a test case always blocks the execution of the control part, i.e., no further test cases can be executed in parallel. In practice, the sequential execution of test cases may be too time-consuming. Hence, one should be able to specify which test cases can be run in parallel. At the same time, there should also be a way to describe dependencies among test cases elegantly.

For that purpose, some ideas from the UNIX tool MAKE can be adopted. An input file for MAKE consists of a number of dependency rules in the form

target : [prerequisite] commands.

A target (typically a file) is achieved by running the corresponding commands. Before these commands can be executed, an optional prerequisite (which may be the target of another dependency rule) must be fulfilled. If the prerequisites of two targets are fulfilled, their commands can be executed in parallel.

For TTCN-3, a new *par* statement is suggested. Its general structure is illustrated by a simple example in figure 3.17. Three goals – *basic*, *cap1*, and *cap2* – are defined within the *par* statement. For each goal, a boolean variable with the same name and the scope of the par operator is defined implicitly. Initially, all variables are set to *false*. If the prerequisite of a goal (provided as a boolean expression in square brackets) is fulfilled, the corresponding statement block is executed. After its termination, the goal variable is automatically set to true. In order to restrict the degree of concurrency, the *par* operator has an optional parameter that specifies the maximum number of parallel test cases (*maxNoOfTestCases* in the given example).

The details of the semantics of the parallel statement – When are the prerequisites tested? What happens if the prerequisite of some goals cannot be fulfilled during execution? – are subject to further studies.

4 Test Generation Based on Formal Specifications

Test generation is the process of deriving a set of test cases from a formal specification.[1] A *specification* is a formal object that prescribes the behavior of a system. A specification is called *formal* if it is defined by means of a *formal description technique* (*FDT*). Formal specifications allow for a non-ambiguous interpretation and thus facilitate verification, validation, and automatic test generation. Formal descriptions techniques include LOTOS, ESTELLE, and the *Specification and Description Language* (*SDL*; see section 5.2).

Automatic test generation based on formal specification provides many advantages over manual test specification:

- *Efficiency of test specification*: Automatic test generation requires little human intervention and thus accelerates the availability of test suites.
- *Correctness of test case*s: The derivation of test cases directly from the specification ensures that the test cases are semantically correct with regard to the specification. In addition, syntactical correctness with regard to the test language is warranted by test generation tools.
- *Effectiveness of test cases*: Coverage analysis techniques that are applied during test generation allow to measure the quality of a test suite.
- *Efficiency of test execution*: Smaller test cases result in faster test execution. Sophisticated test generation algorithms minimize the size of test cases, i.e., the number of test events that are needed to serve a particular test purpose.

In this chapter, some fundamental concepts and techniques of automatic test generation based on formal specifications are presented. In section 4.1, the concepts of conformance testing and test generation are formalized. Several test generation techniques for increasing and assessing the effectiveness of test cases are presented in section 4.2. When dealing with formal specifications, automatic test generation is only one step in the development cycle. The relationship between test generation, verification, and validation is explained in section 4.3.

[1] From a theoretical point of view, it makes no difference whether several test cases or just a single large test case is generated. In practice, of course, a set of small(er) test cases where each test case has its own test purpose is preferable.

4.1 Formal Methods in Conformance Testing

In section 2, ISO/IEC IS 9646 has been introduced as an *informal* framework and methodology for conformance testing. On the other hand, automatic test generation based on FDTs demands for a *formal* interpretation of testing concepts such as conformance, test cases, or test execution. In ITU-T Standard Z.500 *Framework on Formal Methods in Conformance Testing* (*FMCT*; ITU-T, 1997b), their meaning is formalized in terms of mathematical concepts. The main results of FMCT are presented in the following.

4.1.1 Specification and Implementation

The formalization of conformance testing concepts is based on models for both specifications and implementations. In general, a *model* is a representation and an abstraction of anything such as a system, concept, problem, or phenomenon.

A specification s can be considered an element from the set of all possible specifications *SPECS*. Specifications might be parameterized to offer implementation options to the implementor. The set of valid combinations of parameter values can be defined as some set D_s. Then, a parameterized specification s is considered a function that maps from the parameter domain D_s to the set *SPECS* of all instantiated specifications:

$$s : D_s \rightarrow SPECS$$

The supplier of an IUT must specify the chosen implementation options in the implementation conformance statement (ICS). Thus, conformance is always determined based on an instantiated specification.

The set of all implementations is denoted as *IMPS*. In contrast to a specification, an IUT is a *physical* object for which formal reasoning is not possible. However, the general *test assumption* is made that any implementation can be modeled as some $m_{IUT} \in MODS$. *MODS* is a formalism that may be identical to *SPECS*. The purpose of testing is to gain information about the IUT such that m_{IUT} can be constructed in sufficient details. Of course, a given implementation can be modeled by several equivalent models $m \in MODS$. Since these models cannot be distinguished during testing, it is sufficient to consider only one them.

4.1.2 Static and Dynamic Conformance

In order to conform to a given specification, an IUT must meet both static and dynamic conformance requirements.

Static conformance is achieved if the ICS for an IUT defines a valid set of implementation options, i.e., the parameterized specification is instantiated correctly. Formally, this means that $ICS_{IUT} \in D_s$.

Dynamic conformance is established if the observable behavior of the IUT is permitted by the specification. Dynamic conformance is characterized by an *implementation relation*:

$$imp \subseteq MODS \times SPECS$$

An IUT m_{IUT} conforms to a (parameterized) specification s if $m_{IUT}\ imp\ s(ICS_{IUT})$.

The implementation relation *imp* is not fixed. Instead, a large number of "reasonable" implementation relations have been proposed in literature (De Nicola and Hennessy, 1984; Hoare, 1985; Milner, 1989; van Glabbeek, 1993). Some well-known implementation relations are:

- **Trace equivalence**: The set of execution traces of the implementation must be equal to the set of traces of the specification, i.e., the implementation must show exactly the same behavior as the specification.
- **Trace preorder:** The set of execution traces of the implementation must be a subset of the set of traces of the specification, i.e., the implementation is allowed to show only a subset of the behavior of the specification but no additional behavior.
- **Failure preorder:** The set of traces of the specification must be a subset of the set of traces of the implementation and the implementation shall not produce any unspecified deadlocks.

The set of all implementations conforming to an instantiated specification s is given by

$$M_s = \{m \in MODS \mid m\ imp\ s\}$$

If dynamic conformance is defined in terms of a collection of single conformance requirements, a modified formalism is necessary. In that case, an instantiated specification s is expressed as set of requirements $R_s \subseteq REQS$ where $REQS$ denotes the set of all requirements that can be expressed in the requirements language. The set of all possible specifications is defined as $SPECS := \mathbb{P}(REQS)$.

The role of the implementation relation is taken by a satisfaction relation *sat* with

$$sat \subseteq MODS \times REQS$$

An IUT m_{IUT} conforms dynamically to a set of (instantiated) requirements R_s if $\forall r \in R_s : m_{IUT}\ sat\ r$.

4.1.3 Testing Concepts

Testing aims at gaining information about a system in order to be able to decide whether the system has a certain property or not. This is achieved by executing a series of test cases that are formalized in a test notation *TESTS*. Since a test suite ts is a set of test cases, it holds that $ts \subseteq TESTS$ and $ts \in \mathbb{P}(TESTS)$, respectively. It is assumed that the test cases are correctly implemented in the tester.

In a generalized model, the tester does not interact directly with the IUT but with an SUT. The part of the SUT that surrounds the IUT is called the *test context*. The observable behavior of the IUT at its access points may not be identical with the observable behavior of the test context at its PCOs. Thus, the text context is defined as a function

$$C : MODS \rightarrow MODS$$

that maps the model of the IUT to the model of the test context. Then, $C(m_{IUT})$ represents the observable behavior of the IUT at the PCOs.

Execution of test case $t \in TESTS$ means running it in combination with an IUT $m_{IUT} \in MODS$ in a test context C. During execution, an observation $\sigma \in OBS$ is made where OBS denotes the set of all possible observations. Observation σ may include a complete log of all test events and other relevant information. Test execution can be formalized by some function *exec*:

$$exec : TESTS \times MODS \rightarrow OBS$$

For a concrete test case $t \in TESTS$, an implementation modeled by m_{IUT}, and a test context C, $exec(C(m_{IUT}))$ defines the corresponding observation.

For each observation of a given test case t, a *verdict assignment* $verd_t$ is defined:

$$verd_t : OBS \rightarrow \{pass, inconclusive, fail\}$$

An IUT is said to *pass* a correctly implemented test case t if and only if the execution of the test case results in an observation σ to which the verdict *pass* is assigned.

$$\begin{aligned} IUT\ passes\ t &\Leftrightarrow verd_t(\sigma) = pass \\ &\Leftrightarrow verd_t(exec(t, C(m_{IUT}))) = pass \end{aligned}$$

The *test purpose* of a test case t can be formalized as the set of models m for which t results in verdict pass:

$$P_t = \{m \in MODS \mid verd_t(exec_t(t, C(m_{IUT}))) = pass\}$$

In other words, the objective of the test case is to find out whether the model of the IUT belongs to those models that show the correct behavior.

The execution of a (finite) test suite $T \subseteq TESTS$ means executing all $t \in T$. Obviously, an IUT passes a test suite if it passes all of its test cases:

$$\begin{aligned} IUT\ passes\ T &\Leftrightarrow \forall t \in T : IUT\ passes\ t \\ &\Leftrightarrow m_{IUT} \in \bigcap_{t \in T} P_t \end{aligned}$$

The purpose of a test suite is the intersection of the test purposes of all of its test cases:

$$P_T = \bigcap_{t \in T} P_t$$

The quality of a test suite T for some specification s is determined by the relation between its test purpose P_T and the set of all correct implementations M_s:

Relation	Property	Meaning
$P_T \subseteq M_s$	Exhaustiveness	All implementations passing the test suite are compliant to the specification
$P_T \supseteq M_s$	Soundness	All implementations that do not pass the test are not compliant to the specification
$P_T = M_s$	Completeness	The test suite is both exhaustive and sound

Soundness is a fundamental prerequisite for any well-defined test suite, whereas exhaustiveness (and thus completeness) cannot be achieved in general.

4.1.4 Test Generation

Test generation is a process by which a set of test cases is generated from a formal specification. In practice, a high-level *formal description technique* (*FDT*) is used for specification rather than low-level formalisms like petri nets, labeled transition systems, or finite state machines. However, a model in a high-level language is typically transformed into a model in a simpler formalism according to the semantics of the FDT.

Test generation can be modeled as a function *gen* that takes an (instantiated) specification as input and returns a test suite:

$$gen^{C}_{imp} : SPECS \rightarrow \mathbb{P}(TESTS)$$

The generated test suite is dependent on the underlying conformance relation *imp* and the expected test context C of the IUT. A test suite $t = gen^{C}_{imp}(s)$ for a specification s must be sound, i.e., $\forall m \in MODS : m \; imp \; s \Rightarrow m \in P_t$.

If a parameterized specification $s : D_s \rightarrow SPECS$ is given, a parameterized test suite $t : D_s \rightarrow \mathbb{P}(TESTS)$ should be generated that can be applied to implementations with different ICSs. Test generation that operates on parameterized specifications rather than instantiated ones, is modeled by a function *pgen*:

$$pgen^{c}_{imp} : (D_s \rightarrow s) \rightarrow (D_s \rightarrow \mathbb{P}(TESTS))$$

A parameterized test suite $t : D_s \rightarrow \mathbb{P}(TESTS)$ must be sound for all *ICS*, i.e., $\forall ICS \in D_s : t(ICS) \; is \; sound$.

4.2 Test Generation Methods

Except for trivial systems, exhaustive testing requires a very large or even infinite number of test cases. Due to the fact that test execution is subject to time constraints, the amount of test cases must be restricted. This again implies that not necessarily all faults in an IUT might be detected.

When a test suite of reduced size (with respect to an exhaustive test suite) is generated, some criterion is needed to assess its effectiveness. For test generation based on formal

specifications, there are three general approaches for generating and assessing test cases in a systematic way. These are based on fault models, test coverage criteria, and scenario-like requirements. In the following, each approach is described in detail.

4.2.1 Fault Models

A *fault model* is a hypothetical model of what types of faults may occur in an implementation. A fault model can be used for evaluating and optimizing a test suite. It also allows to formalize the test purpose concept.

Formally, a fault model F is a subset of all models of non-conforming implementations (ITU-T, 1997b, p. 24):

$$F \subseteq MODS - M_s$$

Typically, a fault model does not include *all* non-conforming implementations but only those with a particular property. A fault model can also be considered a mutant of specification $s \in SPECS$. For a given modification $\Delta_s \in SPECS$ of s, the corresponding fault model is $M_{\Delta_s} - M_s$.

The extent to which a test suite approximates exhaustiveness with regard to a given fault model F is denoted as *fault coverage*. It allows to quantify the quality of a test suite with regard to the ability to detect errors. Fault coverage is formalized as a function

$$cov_F : \mathbb{P}(TESTS) \rightarrow [0, 1]$$

that maps a test suite into a real number between 0 (no coverage) and 1 (full coverage).

While the exact definition of cov_F depends on a concrete fault model F, a general requirement is that the coverage must increase if more faulty implementations in F are detected. Given two test suites T_1 and T_2, the following condition must hold

$$F - P_{T_1} \subseteq F - P_{T_2} \Rightarrow cov_F(T_1) \leq cov_F(T_2)$$

where P_{T_i} is the formal test purpose of test suite T_i, i.e., the set of all implementation models that pass the test suite, and $F - P_{T_i}$ is the set of models of implementations that fail with T_i.

Fault models are defined dependent on the formalism of the specification or the implementation model. For instance, if the specification is defined in SDL (see next chapter), one possible fault model is that messages are sent to the wrong receiver. Typical faults of software specified in an imperative programming language are missing alternatives, incorrect operators (e.g., $\leq$ instead of $<$), negation of boolean expressions in an `if` statement, improper block structuring, and missing or wrong initialization of variables.

Test Generation Based on Finite State Machines

A formalism for which many test generation methods have been developed are finite state machines (FSMs).

Definition 1 (Deterministic Finite State Machine) *A deterministic finite state machine (FSM) is a 6-tuple* $FSM = \langle S, I, O, \sigma, \varphi, s_o \rangle$ *where*

- $S = \{s_1, \ldots, s_n\}$ *is a finite set of states*
- $I = \{i_1, \ldots, i_k\}$ *is a finite set of inputs*
- $O = \{o_1, \ldots, o_l\}$ *is a finite set of outputs*
- $\sigma : S \times I \rightarrow S$ *is a transition function that determines the next state after some input*
- $\varphi : S \times I \rightarrow O$ *is a function that determines the output for every transition*

An FSM can be represented as a graph $G = \langle N, E \rangle$ *where* $N = S$ *is a finite set of nodes and* $E = S \times (I \times O) \times S = \{(s, (i, o), s') | \sigma(s, i) = s' \wedge \varphi(s, i) = o\}$ *is a finite set of directed and labeled edges.*

With regard to FSMs, two fault models can be identified:

- A *transfer fault* occurs when a state transition drives the implementation in a wrong state, i.e., σ is implemented incorrectly.
- An *output fault* occurs when an IUT responds with a wrong output in a given state and for a given input, i.e., function φ is implemented incorrectly.

In order to create test cases for both fault models, several algorithms have been proposed that are sketched in the following. They are all based on the following three assumptions:

1. The FSM describing the specification is strongly connected[2], i.e., all states are reachable.
2. The FSM describing the specification is minimized, i.e., there are no states that are strongly equivalent[3].
3. The FSM describing the implementation has at least as many states as the FSM of the specification.

The third assumption is essential, because otherwise an FSM can be constructed that passes all tests by simply using as many states as transitions in the test sequence.

Transition Tours. The *transition tour* method (Naito and Tsunoyama, 1981) aims at generating test sequences that cover all transitions of the FSM at least once. A transition tour is suitable for checking whether the implementation produces incorrect outputs. But it does not check whether the FSM of the implementation is in the right state after each input (transfer fault).

[2] An FSM is strongly connected if for any two states s_i and s_j, there exists an input sequence such that s_j is reachable from s_i.

[3] Two states s_i and s_j are strongly equivalent if any input sequence results in identical output sequences for s_i and s_j.

Distinguishing Sequences. A *distinguishing sequence* (*DS*) is a sequence of inputs that produces different outputs for each state of a given FSM. This means that for two starting states s_1 and s_2 ($s_1 \neq s_2$), the distinguishing sequence $ds = i_1, \ldots i_n$ will make the FSM produce different outputs. If a distinguishing sequence can be determined for the FSM of the specification, a two-step algorithm suggested by Kohavi (1978) can be applied:

In the *state verification* phase, the response of the implementation to the distinguishing sequence is retrieved for each of its states. This is achieved in three steps:

1. The IUT is reset, i.e., it is set back to the initial state.
2. A fixed preamble $p(i)$ is applied that is supposed to drive the implementation into a state p_i which is isomorphic to state s_i of the implementation.
3. The DS is applied to the implementation and the output is checked for equality with the output of the specification if the DS is applied in state s_i.

State verification ensures that the implementation has at least as many states as the specification. Nevertheless, there might be transition faults when executing both the preambles and the DS.

Thus, in the *transition verification* step, each single transition $t = \langle s_i, (i, o), s_j \rangle$ is tested. This is achieved in three phases:

1. The implementation is put into state p_i which is DS-isomorphic to s_i by resetting the implementation and applying preamble $p(i)$.
2. Input i is applied to the implementation and the actual output is compared with the expected output o.
3. The new state p_j of the implementation is checked to be DS-isomorphic to s_j by applying the distinguishing sequence to p_j and s_j and checking the resulting output for equality.

The W Method. An FSM does not necessarily have a distinguishing sequence and thus the test method above cannot be applied in all cases. The *W method* (Vasilevskii, 1973; Chow, 1978) solves this problem by considering a set $W = \{w_1, \ldots, w_n\}$ of input sequences. For any state s_i, the application of all $w_j \in W$ uniquely identifies s_i.

The W method follows the general approach of the DS algorithm. In the state verification phase, the response of the implementation to all sequences in W is retrieved for each state. As a consequence, the IUT must be reset $|W|$ times for each state. In the transition verification phase, the procedure for distinguishing sequences is repeated for each $w_i \in W$.

Unique Input Output Sequences. The former test generation methods tend to produce very long test cases. Thus, Sabnani and Dahbura (1988) have introduced the concept of *unique input/output* (*UIO*) sequences. A UIO for some state s_i, denoted by $UIO(i)$, is

a sequence of inputs and outputs that uniquely identify s_i. It can be proven that if a state has a UIO, it is at most as long as a distinguishing sequence (or W set) common for all states; in practice, they are much shorter.

To prove that all UIOs are indeed unique for their corresponding state, in the state verification phase it is checked that they are rejected for all other states. For all i and j with $i \neq j$ the following procedure is applied:

1. The implementation is put into state p_i which is UIO-isomorphic to s_i by resetting the implementation and applying preamble $p(i)$.
2. $UIO(j)$ is applied to the implementation and the actual output is checked whether it is *not* identical to the expected output of $UIO(j)$ applied to state s_j.

In the transition verification phase, all transitions are tested in a way similar to the DS method. Various optimizations to the UIO method as well as solutions for the case that a state does not have a UIO have been proposed. An overview is given by Cavalli and Anido (1997).

The above-mentioned test generation methods have had a strong influence in the past, but their applicability is restricted for two reasons. First, today's specification have become too large to compute exhaustive test cases within a reasonable period of time. Second, they are based on deterministic FSMs and thus only consider the control flow of deterministic specifications.

For most modern protocols this is no longer adequate and extended FSMs (EFSMs) and their variants which support variables and conditional transitions are a more suitable. Obviously, simplifying an EFSM into an FSM by ignoring all internal variables, input parameters, and enabling conditions is not appropriate.[4] Bourhfir et al. (1997) present a test generation algorithm for EFSMs that combines the UIO approach with the *all-du paths* coverage criterion (see section 4.2.2.2) to take into account both control flow and data flow aspects.

4.2.2 Test Coverage Criteria

The *coverage* of a system is a measure for the extend to which a set of test cases is complete. Test coverage is always related to a particular test method[5]. In the following subsections, different test methods based on control flow and data flow are presented.

Normally, test coverage is determined for white box tests where the control/data flow of IUT itself is subject to investigation. In the context of automatic test generation, the test coverage can only be measured with regard to the control/data flow of the specification, although the tests are applied to the IUT. Since the control and data flow of the specification and the IUT differ, no statements can be made about the coverage of

[4] In addition, many generated test cases will be infeasible.

[5] In this context, the term *test method* refers to the choice of test cases rather than the test architecture as in section 2.4.

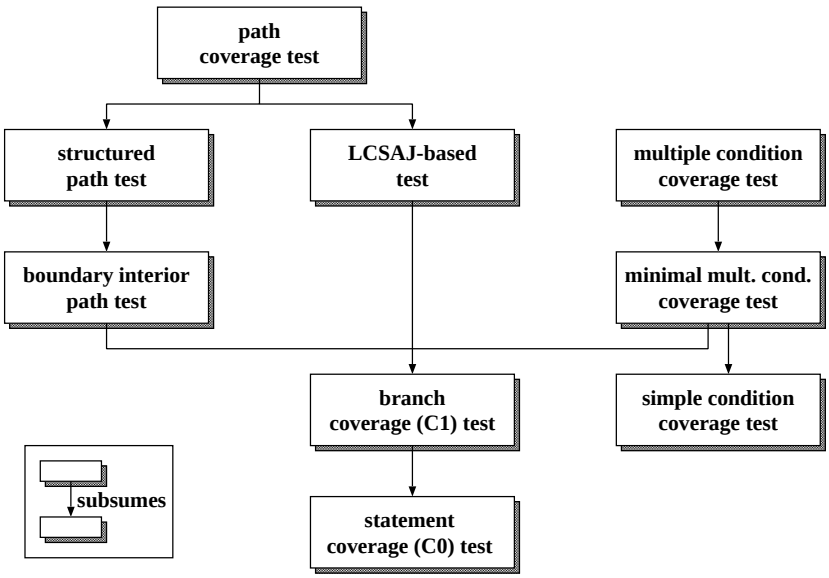


Figure 4.1: Test methods based on control flow

the IUT, i.e., a test suite with a total coverage of the formal specification may leave large parts of the IUT untouched. On the other hand, considering test coverage for automatic test generation can be valuable to ensure that all system requirements (which are given in the specification) are actually tested.

4.2.2.1 Control Flow Criteria

Test methods based on the control of a program or specification consider structuring elements such as statements, branches, and conditions to define test cases. A program is modeled by a control flow graph where each node corresponds to a program statement and the directed edges define the control flow (possibly constrained by conditions). Depending on the requirements on testing (e.g., thoroughness), different test methods can be applied. A hierarchy of control flow-based test methods is shown in figure 4.1.

Statement and Branch Coverage Tests. A *statement coverage test* – also called *C0 test* – aims at executing all statements of a given program or specification at least once. The actual coverage is computed by $\frac{\text{Number of executed statements}}{\text{Total number of statements}}$.

A *branch coverage test* (*C1 test*) requires that all edges of the control flow graphs are executed at least once. That means, for every statement with a condition, tests for all possible alternatives (e.g., *true* and *false* in case of a boolean condition) must be defined. Total branch coverage implies total statement coverage. Branch coverage is considered the minimal test criteria in order to place confidence in an implementation.

Path Coverage Tests. A branch coverage test is not sufficient for a profound analysis of loops because each loop has to be executed only once (hence named C*1* test). Moreover, it does not consider the dependencies between single branches. The *path test* faces these problems by demanding that *all* possible paths must be executed. If a program has two consecutive conditional statements, a path test requires to examine all four possible paths (with boolean conditions evaluating to *true*/*true*, *true*/*false*, *false*/*true*, and *false*/*false*), whereas only two paths (e.g., *true*/*true*, *false*/*false*) must executed for a branch coverage tests.

Due to the large number of possible paths[6] for non-trivial programs, *path tests* have no practical relevance and thus one of the weaker path coverage tests must be chosen. The *boundary interior path test* considers all paths except those for which a loop has to be executed more than once. A generalization of this test criteria is the *structured path test* which ignores only those paths for which an innermost loop (i.e., a loop that contains no other loop) is not executed more than k times (Ck *test*).

For *LCSAJ* (*Linear Code Sequence And Jump*) *tests*, each linear sequence of executable statements and the target to which control flow is transferred at the end of the sequence is considered. This test criteria has been designed for programs with many jumps and thus can be ignored for modern (structured) programs.

Condition Coverage Tests. If a program contains statements with complex conditions, even a path test might be insufficient. Given the statement

$$\text{if } (((x \geq 0) \wedge (x \leq 100)) \wedge (x \bmod 2 = 0)) \ \{\ldots\} \text{ else } \{\ldots\}$$

a path test is only concerned with the condition evaluating to true or false *as a whole* such that both statement blocks can be executed. In contrast, a *condition coverage test* regards each atomic condition, i.e., $x \geq 0$, $x \leq 100$, and $x \bmod 2 = 0$, separately.

Three types of condition coverage tests are distinguished: For a *simple condition coverage test*, all atomic conditions in a program must evaluate both to true and false at least once. This simple test method does not guarantee that all statements are covered. If x is set to -1 and 101 in the given example, all three atomic conditions once evaluate to true and false. On the other, the composed condition always evaluates to false such that the statement block of the `if` statement is never executed. For that reason, simple condition coverage tests should always be combined with other test methods.

Multiple condition coverage tests aim at testing all possible combinations of truth values for the atomic conditions. A condition with n atomic boolean expressions will result in 2^n combinations. In practice, not all of these combinations are feasible. For example, there is no variable assignment that makes both atomic conditions in $((x = 1) \vee (x = 2))$ evaluate to true. The *minimal multiple condition coverage test* avoids this problem by only demanding that all atomic as well as all composed conditions must evaluate to true and false at least once. If a condition is structured hierarchically (where the structure

[6] In most cases, the number of paths is only limited because the range of data types and the size of buffers etc. is restricted. (Due to limited memory, a computer can take only a finite number of states.)

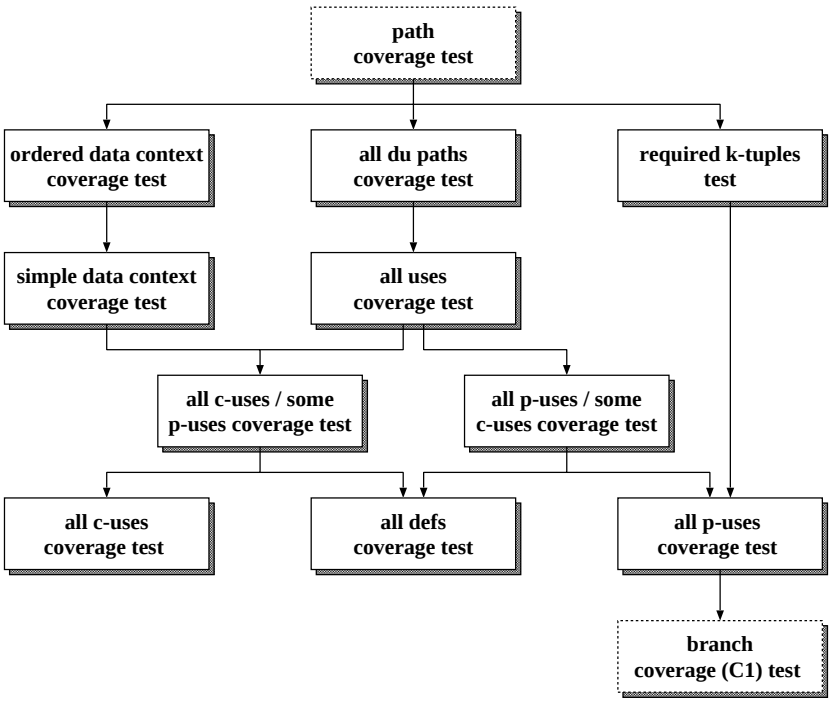


Figure 4.2: Test methods based on data flow

is determined by bracketing and operator precedence), all subconditions must evaluate to true and false as well. For condition

$$(((x \geq 0) \wedge (x \leq 100)) \wedge (x \bmod 2 = 1))$$

the whole expression as well as its substructures $(x \geq 0) \wedge (x \leq 100)$, $x \geq 0$, $x \leq 100$, and $x \bmod 2 = 0$ must be considered.

4.2.2.2 Data Flow Criteria

Data flow-oriented test methods analyze the usage of program variables to define a set of test cases. A hierarchy of data flow test criteria and their relation to methods based on control flow (printed in dotted boxes) is shown in figure 4.2.

Every access to some variable v can be placed in one of the following categories:

- *Definition* (*def*): A new value is assigned to v, i.e., the variable appears either on the left-hand side of an assignment statement or in an input statement.
- *Computational use* (*c-use*): The variable is used within an expression on the right-hand side of an assignment statement or in an output statement.
- *Predicate use (p-use)*: The variable affects the control flow, i.e., it is used within a condition.

The statement $y := x*2+5$ contains a computational use of x, denoted by *c-use*(x), and a definition of y (*def*(y)). In statement if $(x! = 0)$ $\{\ldots\}$, a predicate use of x (*p-use*(x)) is made.

For any program or specification, the data flow can be described by a data flow graph (DFG). A DFG is graph $G = \langle N, E\rangle$ where the finite set of nodes N represents the functional units of a program (single statements, sequences of statements, or even procedures) and the set of directed edges $E = N \times N$ represents the flow of data objects. Definitions and computational uses only occur in the nodes of the DFG while predicate uses only appear in its edges.

A common property of data flow-based test methods is that they consider the impact of the definition of variables on their successive use. Therefore, paths from a variable definition to all computation uses and/or predicate uses are examined.

Definition 2 (Global definitions and c-uses) *A variable definition def*(v) *is called* global *if it is not followed by a c-use*(v) *within the same node. A c-use*(v) *for some variable* v *in a node* $n \in N$ *of the DFG is called* global *if there is no previous variable definition def*(v) *within the same node.*

For each node n_i*, def*(n_i) *denotes the set of globally defined variables and c-use*(n_i) *denotes the set of variables for which there is a global computation use. p-use*(n_i, n_j) *denotes the set of variables with predicate use at edge* (n_i, n_j)*.*

Definition 3 (*dcu* and *dpu*) *For any nodes* $n_d, n_c, n_p \in N$ *and any variable* v*, the set of paths from a definition to a computational/predicate use is defined as follows:*

$$\textit{dcu-path}(n_d, v, n_c) := \begin{cases} \{\langle n_1, \ldots, n_l\rangle \mid n_1 = n_d \wedge n_l = n_c \wedge & v \in \textit{def}(n_d) \wedge \\ \quad \forall i, 1 \le i \le l-1 : (n_i, n_{i+1}) \in E \wedge & \quad v \in \textit{c-use}(n_c) \\ \quad \forall i, 2 \le i \le l-1 : v \notin \textit{def}(n_i)\} & \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\textit{dpu-path}(n_d, v, n_p) := \begin{cases} \{\langle n_1, \ldots, n_l\rangle \mid n_1 = n_d \wedge n_l = n_p \wedge & v \in \textit{def}(n_d) \wedge \\ \quad \forall i, 1 \le i \le l-1 : (n_i, n_{i+1}) \in E \wedge & \quad v \in \textit{p-use}(n_{l-1}, n_p) \\ \quad \forall i, 2 \le i \le l-1 : v \notin \textit{def}(n_i)\} & \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\textit{dcu-path}(n_d, v) := \bigcup_{n_p \in N} \textit{dcu-path}(n_d, v, n_c)$$

$$\textit{dpu-path}(n_d, v) := \bigcup_{n_p \in N} \textit{dpu-path}(n_d, v, n_p)$$

$$\textit{dcu}(n_d, v) := \{n_c \in N \mid \textit{dpu-path}(n_d, v, n_c) \neq \emptyset\}$$

$$\textit{dpu}(n_d, v) := \{n_p \in N \mid \textit{dpu-path}(n_d, v, n_p) \neq \emptyset\}$$

If a variable v is defined globally at node n_d and used computationally in node n_c, then $dcu(n_d, v, n_c)$ denotes the set of all paths from n_d to n_c such that v is not re-defined in between. This means the definition at node d_d has an impact on the computation in node n_c.

Based on the former definitions, seven coverage criteria can be defined:

- *all defs*: All variable definitions must be tested with either a computational or predicate use, i.e., for all $n_d \in N$ and for all $v \in def(n_d)$, at least one path in $dcu\text{-}path(n_d, v) \cup dpu\text{-}path(n_d, v)$ must executed.
- *all c-uses*: All variable definitions must be tested with all successive computational uses, i.e., for all $n_d \in N$, for all $v \in def(n_d)$, and for all $n_c \in dcu(n_d, v)$ at least one path in $dcu\text{-}path(n_d, v, n_c)$ must be executed.
- *all p-uses*: All variable definitions must be tested with all successive predicate uses, i.e., for all $n_d \in N$, for all $v \in def(n_d)$, and for all $n_p \in dpu(n_d, v)$, at least one path in $dpu\text{-}path(n_d, v, n_p)$ must be executed.
- *all c-uses/some p-uses*: In addition to the *all c-uses* criteria, a variable definition must be tested with a predicate use in case there is no successive computational use. I.e., for all $n_d \in N$ and for all $v \in def(n_d)$, if $dcu(n_d, v) \neq \emptyset$, then for all $n_c \in dcu(n_d, v)$ at least one path in $dcu\text{-}path(n_d, v, n_c)$ must be executed; otherwise one path in $dpu\text{-}path(n_d, v)$ must be executed.
- *all p-uses/some c-uses*: In addition to the *all p-uses* criteria, a variable definition must be tested with a computational use in case there is no successive predicate use. I.e., for all $n_d \in N$ and for all $v \in def(n_d)$, if $dpu(n_d, v) \neq \emptyset$, then for all $n_p \in dpu(n_p, v)$ at least one path in $dpu\text{-}path(n_d, v, n_p)$ must be executed; otherwise one path in $dcu\text{-}path(n_d, v)$ must be executed.
- *all uses*: Both the *all c-uses* and the *all p-uses* criteria must be met.
- *all du paths*: All variable definitions must be tested with all successive computational and predicate uses; in contrast to the *all uses* criterion, *all* paths from a definition to a variable use that are cycle-free or simple-cycles must be executed. I.e., all cycle-free and simple-cyclic paths in $\bigcup_{n_d \in N} \bigcup_v (dcu\text{-}path(n_d, v) \cup dpu\text{-}path(n_d, v))$ must be executed.

Studies by Girgis and Woodward (1986) have shown that the *all c-uses* criteria detects more errors than the *all p-uses* criteria.

Simple and Ordered Data Context Coverage Test. The underlying idea of the *simple* and *ordered data context coverage* test is that if a variable is used in a statement, than at least one path from every possible preceeding definition to that statement must be tested. If more than one variable is used in a statement, than a path must be chosen for each possible combination of preceeding definitions.

Let $d_n(v)$ denote the definition of variable v at node n. $d_n(v)$ is said to be *live* at statement m if there is a definition-free path from n to m with regard to v. An *elementary*

context of statement s consists of the definitions that are live at statement s for a particular path to statement s. The *simple data context* $DC(i)$ of statement s is defined as the set of all of its elementary contexts. The *ordered data context* $ODC(i)$ also takes into account the order in which definitions occur.

Example: For some statement $x := y + z$ in state s_6, all possible combinations of definitions that are live in s_6 are described by

$$DC(6) = \{\langle d_1(x), d_4(y)\rangle, \langle d_1(y), d_3(x)\rangle, \langle d_3(x), d_4(y)\rangle\}$$

The simple data context coverage test requires that paths for all elements in $DC(i)$ must be tested. For the first element of $DC(6)$, a paths covering the three states 1, 4, and 6 must be tested. In some cases the definitions can be executed in varying order. If both $\langle d_i(x), d_j(y)\rangle$ and $\langle d_j(y), d_i(x)\rangle$ are part of an ordered data context, the ordered data context coverage test requires that two paths must be tested.

4.2.3 Scenario-Based Requirements

Test generation based on fault models and coverage criteria allows for thorough testing but the number and size of the tests generated may be too high for practical application. On the other hand, a test specifier often has a clear notion of what functionality must be tested to gain confidence in an implementation.

In principle, there are two kinds of specifications:

- *Scenario-based specifications* describe the behavior by a (possibly incomplete) set of single traces.
- *System-based specifications* describe all possible behavior by a single model.

In scenario-based specifications, each trace (scenario) describes one (or several) particular requirement(s) of the system. Thus, such scenarios can be interpreted as test purposes. In contrast, a system-based specification describes a (possibly infinite) set of traces.

Scenario-based requirements can be described in various ways, ranging from basic (temporal) logics such as the *computation tree logic* (*CTL*; Clarke and Emerson, 1981) and automata-based models (see observer processes in section 6.2.3) to high-level and more user-friendly notations such as *Message Sequence Chart* (*MSC*; see section 5.1).

The general distinction between specifications and implementations mainly depends on the level of abstraction and the stage in the development process. System-based specifications often abstract from implementation issues. On the other hand, a very detailed system-based specification may be considered a reference implementation.

For test generation purposes, it is admissive to consider single scenarios as requirement specifications on a system-based specification. Thus, a pragmatic approach is to generate a test case for each of these requirements. In this context, test generation means to complement the traces with regard to events resulting in *inconclusive* and *fail* verdicts,

and to transform them into another representation in a desired test language.[7] A scenario may be incomplete in so far as it does not describe all observable events that are necessary to drive a system from an idle state to the same or another idle state. For instance, the test specifier may leave out the preamble and postamble. In that case, a complete trace must be determined that subsumes the given one.

Very often, at least informal scenarios are defined prior to a system-based specification by a protocol or system designer. If no scenarios are available, the task to define them is left to the test specifier. By simulation techniques, the test specifier is able to derive single scenarios interactively from a system-based specification.

4.3 Test Generation, Verification, and Validation

Test generation based on formal specifications is not an isolated task but often intertwined with other activities. Two analysis methods for formal requirements and system specifications are *verification* and *validation*.

- **Verification** is the process of evaluating a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase (IEEE, 1990). Verification ensures that a model is transformed from one form into another with sufficient accuracy. It aims at answering the question "Have we built the *system right*?"
- **Validation** determines the correctness of the products of software development with respect to the user needs and requirements (IEEE, 1990). Validation is regarded as an informal process, because the requirements are only in the user's mind (Kneuper, 1992). It aims at answering the question "Have we built the *right system?*"

The terms *verification* and *validation* are often used as synonyms or even get mixed up in literature. Some authors consider verification as a static method based on formal proofs while validation involves dynamic execution of a model.

In fact, verification of a model can either follow an axiomatic or a simulation approach. In the latter case, a *model checker* determines whether the reachability state graph contains paths that exhibit a specific property. There exist three types of properties that differ in the number of paths that must exhibit the property:

Property Class	**Paths**
Safety	none
Liveness	some
Invariance	all

A invariance property is the logical complement of a safety complement. To check a model for safety or invariance properties, the simulation must be exhaustive. Model

[7] Of course, these procedures are also necessary for fault-based and coverage-based test generation.

checkers employ various methods to reduce the complexity of search (see section 8) but for complex models the state space is nevertheless infinite. Thus, model checkers cannot prove safety and invariance properties but only refute them by providing a counter-example.

Verification and validation tools such as the TAU SDL VALIDATOR (Telelogic, 2001) provide a number of predefined general and language specific rules that are checked during simulation. In addition, the user may specify his own individual rules. Whenever a rule is satisfied during state space exploration (i.e., a liveness property is proven or a safety/invariance property is refuted), a report with the current path through the reachability graph is stored for later investigation. For parallel systems, undesired behavior such as deadlocks and livelocks can be detected. If communication in a distributed system is based on message exchange, it can be checked whether the receiver specified in an output statement exists and is reachable by the sender at run-time. Further rules check for arithmetic and array operations such as divisions by zero or out-of-bounds accesses.

A Process Model. Validation, verification and test generation are closely related to each other, in particular in the context of scenario-based test generation. All three techniques are based on the dynamic execution of a system-based specification. By validating the system, traces are generated that can be used as input for scenario-based test generation. During test generation, a given scenario is verified implicitly as a liveness property of the system. If there is no path in the reachability graph that matches with the scenario, the system-based specification and the scenario are contradictory and no test case can be generated.

In figure 4.3, a process model for specification, validation, verification, and test generation is presented. Typically, the development process starts by stating the system requirements by means of simplified scenarios. Based on these scenarios, a complete system specification based on another formalism is derived. This system specification is subject to validation, verification, and test generation.

If validation uncovers a fault, the system specification must be modified. By verification, the system specification is checked against the system requirements. Model checking may fail due to various reasons: If the requirement is violated by the system specification, the latter must be modified. There are, however, situations, where the system requirements themselves turn out to be improper, e.g., because some technical aspects were not taken into account when they were specified or because they are too informal or imprecise for automatic model checking. In this case, the original system requirements must be modified or refined.

Only after validation and verification of the system specification have been completed successfully, i.e., the system specification is considered “stable”, automatic test generation is performed. The resulting test suite is validated by the test specifier. If a fault is detected, either the system requirements or the system specification must be adapted.

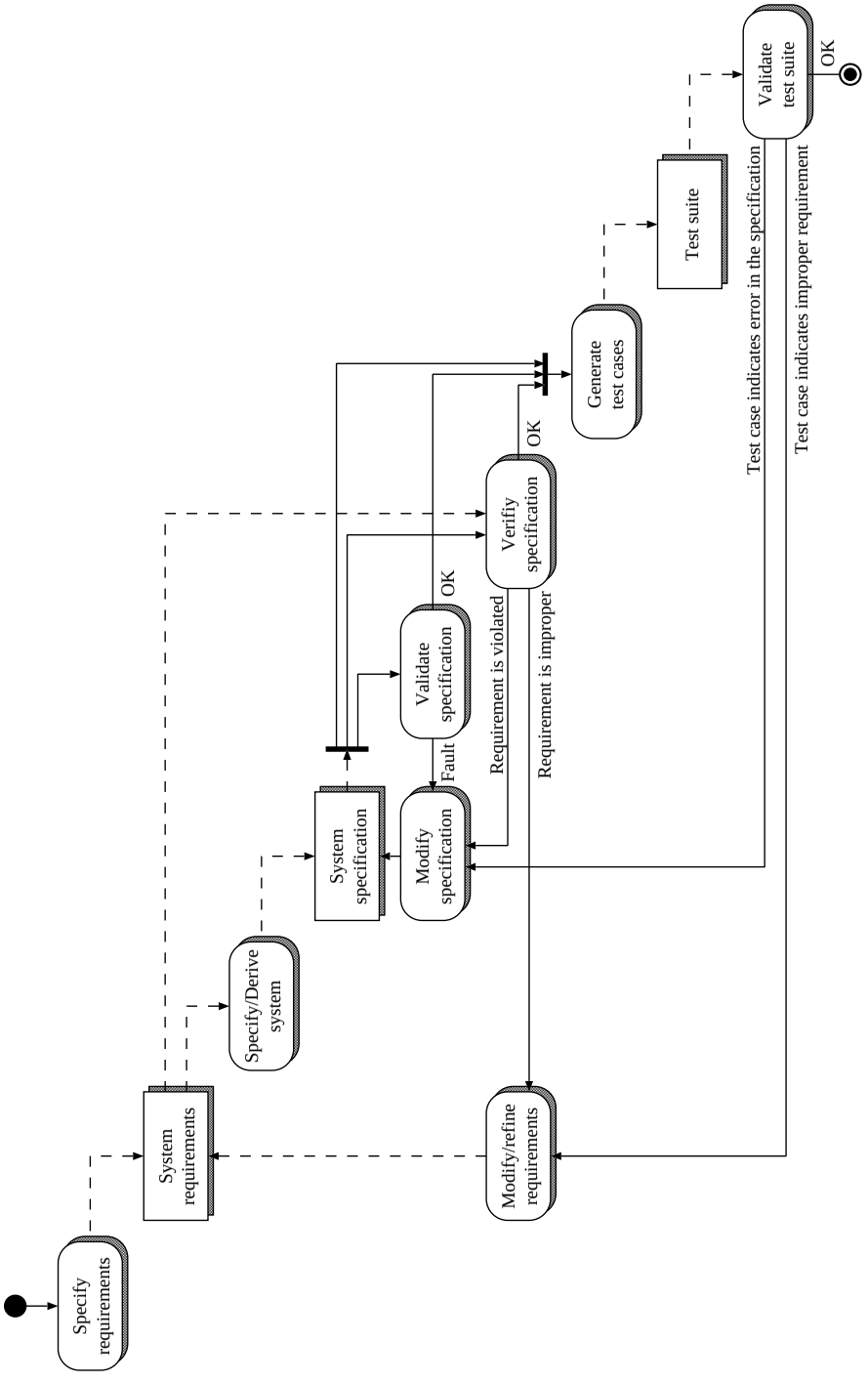


Figure 4.3: A process model for specification, validation, verification, and test generation

5 High-Level Specification Languages

The necessity to apply modeling methods for the development of software and technical systems is undisputed as the complexity of these systems is increasing continuously. Requirement and system specification not only presupposes appropriate basic concepts, such as asynchronous process communication, but also standardized high-level languages.

In the telecommunication area, *Message Sequence Chart* (*MSC*, see ITU-T 1999d; 1996b) and the *Specification and Description Language* (*SDL*, ITU-T 1999a; 1996a; 1992) have become the dominating modeling languages. MSC allows to specify the interaction between a number of independent instances by describing single scenarios. It is typically used during requirement analysis but it can also be applied for testing. On the other hand, SDL is used for the specification of complete systems. It has concepts for describing both behavior, data, and structuring. SDL is applied in the design and implementation phase.

MSC and SDL have many properties in common that suggest a combined use. First of all, both languages have a graphical and a textual notation. While the phrase representation is mainly intended for exchange between tools, the graphical format allows to present information about structure or order of events in an intuitive way. Secondly, SDL and MSC are formal languages, i.e., a formal semantics is available that allows to interpret a given specification in an unambiguous way. This feature distinguishes SDL and MSC from many other graphical languages used in software engineering, e.g., the *Unified Modeling Language* (*UML*; see Rumbaugh et al., 1999; Object Management Group, 1999). Even though SDL and MSC can be used for sequential programs, their main focus is on the description of distributed systems whose components communicate by asynchronous message passing.

In the following, a short introduction to the main language concepts of MSC and SDL is given.[1] Their application is illustrated by examples based on the Inres protocol.

5.1 Message Sequence Chart

Message Sequence Chart is a graphical specification language standardized by the ITU-T as Recommendation Z.120. MSC can be used for describing the communication behavior among system components and their environment where communication is realized by

[1] A modified version of the introduction to SDL has been published as Grabowski et al. (2002).

asynchronous message exchange. Each MSC diagram[2] depicts one or more traces of the system.

MSC provides three types of diagrams: A *(basic) MSC* describes the concrete events that take place at the various system components and their temporal ordering. Possible events include the sending and receiving of messages, local actions, timer operations and instance creation/termination. A *High-level MSC* (*HMSC*) abstracts from system components but provides a road map of how to combine sets of MSCs. HMSCs allow to arrange MSCs sequentially, as alternatives, and in parallel. Finally, an *MSC document* provides a kind of table of contents. It contains general information, e.g., it lists all MSC diagrams that belong to a project, declares the instances and data used within these diagrams, and refers to related documents. In practice, MSC documents are used seldomly. In the following, the main features of basic MSCs and HMSCs will be discussed in detail.

5.1.1 Basic Message Sequence Charts

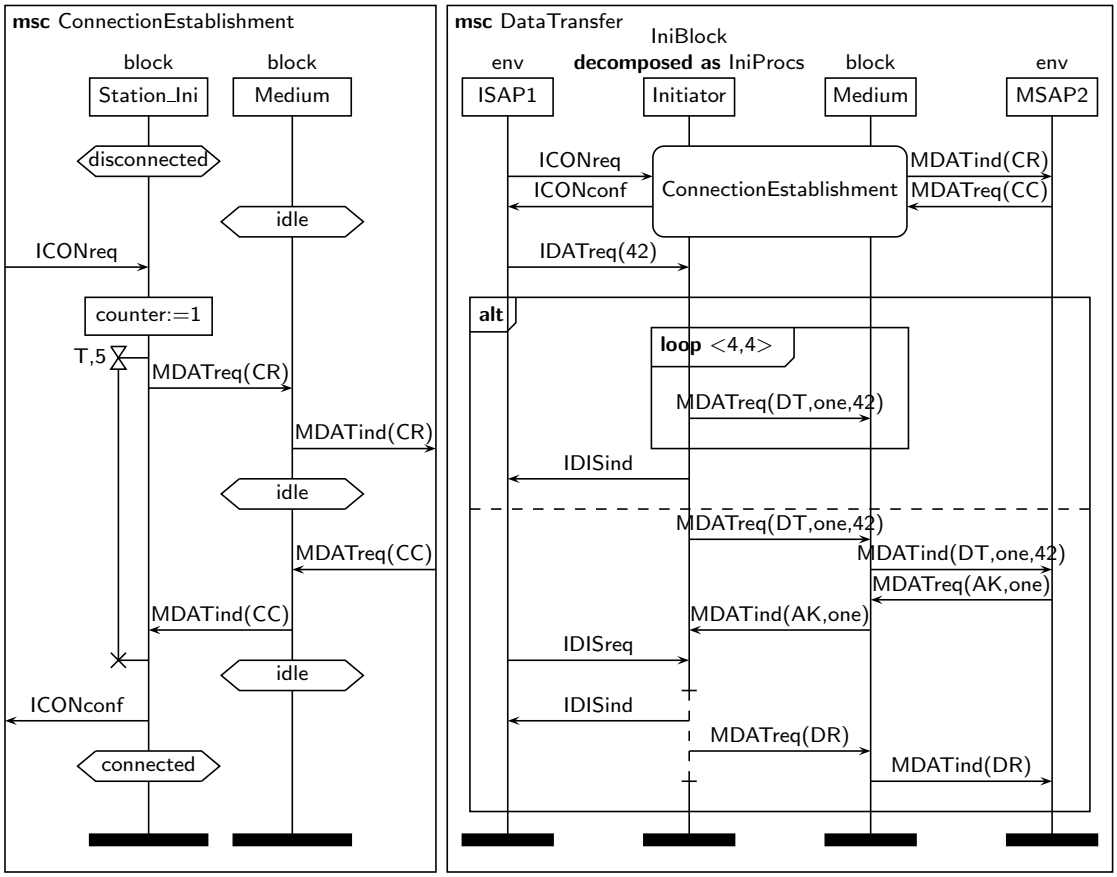
Instances, Messages, and Control Flow. The main language concepts of MSC are *instance* and *message*. Instances represent components that exchange messages asynchronously with each other and with the system environment. An instance has a name and an optional type.

A message is characterized by its name and an optional number of parameters. A message exchange defines two events: The sending/output and the reception/input of the message. Typically, a message involves two instances (components) or one instance and the environment. However, there also exist special symbols in order to describe messages that are lost or come from an unknown sender.

Figure 5.1(a) shows an MSC with two instances, *Station_Ini* and *Medium*, that are displayed as vertical lines with an additional rectangle for the instance header and a horizontal bar for denoting the instance end. Instance *Station_Ini* is a block that represents an Inres protocol instance at the sender (initiator) side. Instance *Medium* represents an underlying medium over which data are exchanged with some imaginary responder. The MSC describes the typical scenario of a connection establishment: If *Station_Ini* receives message *ICONreq* (represented by the annotated arrow pointing from the diagram border to the instance axis), the connection request is forwarded via the medium (messages *MDATreq(CR)* and *MDATind(CR)*). Provided that the other party responds with a confirmation (*MDATreq(CC)* and *MDATind(CC)*), a confirmation message (*ICONconf*) is sent by *Station_Ini*. Typically, the description of an instance finishes with a special end symbol. However, this symbol does not mean that the instance actually terminates.

MSC defines a total ordering along each separate instance axis where each instance has to be interpreted from top to bottom. Events on different instances are only partially

[2] In the following, the term *MSC* may denote both the MSC language as well as a single MSC diagram. An explicit linguistic distinction between MSC language and MSC diagram is only made when there is an ambiguity otherwise.



(a) Basic MSC (b) MSC Expressions, Coregions, and Gates

Figure 5.1: Basic MSCs for the Inres protocol

ordered by message exchange in the sense that a message has to be sent before it can be received. For that reason, it is semantically irrelevant whether a message arrow points upwards or downwards even though the latter representation is supposed to be more intuitive for the viewer.

Due to the partial ordering, an MSC often represents not just one trace but a set of (similar) traces. In some cases, it is even too restrictive to have the events along one instance ordered totally. For example, if an instance receives messages from different sources, the order in which the messages arrive may be unpredictable and irrelevant. MSC allows to mark parts of an instance as a *coregion* in which all events are unordered. In figure 5.1(b), a coregion (represented by a dashed line) has been specified for instance *Station_Ini*, since the order in which *IDISind* and *MDATreq(DR)* are emitted is irrelevant and hence should be left unspecified.

Besides asynchronous message exchange, MSC supports method calls and replies as another communication mechanism. Method calls may be either asynchronous or synchronous. In the first case, the caller can proceed execution without waiting for the reply, whereas a synchronous call put the caller in a suspension mode until the reply of the method arrives.

Environment and Gates. Each MSC diagram is delimited by a rectangular frame that represents the *environment*. Instances are allowed to communicate with this environment by message exchange. Each message input or output event that is associated with the environment is assigned to a *gate* whose name is given either explicitly or implicitly by the corresponding message. Gates are a kind of interface that allows to compose several MSCs in a vertical manner, i.e., a message may be sent from some instance A to a gate in one diagram and sent from the same gate to another instance B in a second diagram. An example for re-using MSCs with gates is given by MSC *DataTransfer* (figure 5.1(b)) that refers to MSC *ConnectionEstablishment*.

In contrast to instances, the environment as a whole as well as all input and output events belonging to the same gate are not totally ordered. Therefore, if the gate concept is not needed for combining MSCs, in many cases it is better to represent the system environment by one or more additional instances as done in figure 5.1(b).

Actions. In addition to message communication, the internal actions of an instance can be described in an MSC. An action is represented by a rectangular symbol and may either contain a formal statement or an informal text. In figure 5.1(a), instance *Initiator* has an action box in which variable *counter* is set to *1*.

Conditions. MSC provides two kinds of conditions: A *setting* condition defines the current state of one or more instances where the state is described by a name. In contrast, *guarding* conditions restrict the behavior of instances by making the execution of subsequent events dependent on a boolean expression or a state which the instance should be in. Guarding conditions are typically used in combination with alternatives.

In figure 5.1(a), the states of instances *Station_Ini* and *Medium* are shown by five (setting) conditions which are represented as hexagons. After a successful connection establishment, *Station_Ini* changes from state *disconnected* to *connect*.

Conditions may not only be associated to a single instance but can also refer to several or even all instances of an MSC. A setting condition that involves all instances describes a current global system state.

Timers and Time Constraints. The description of timers is supported by three events, namely *start*, *timeout* and *stop*. In MSC, a timer is associated with exactly one instance, i.e., there is no notion of a global timer that is accessible by all instances (though MSC assumes a global clock). Hence, a start event and a corresponding timeout/stop event have to be specified at the same instance axis. In MSC *ConnectionEstablishment* (figure 5.1(a)), timer T guards the reception of a connection confirmation. In the given scenario, message *MDATind(CC)* arrives in time and the timer is stopped.

MSC-2000, the latest revision of the standard, provides extensive support for the description of real-time systems. MSC constructs can be annotated with *time constraints* that (a) specify the point in time of single events (absolute timing) or (b) define a time

interval between two events (relative timing). Time constraints can be defined as exact time points or as time intervals with upper and lower bounds.

Both relative *time measurements* (which observe the value of a global clock) and absolute time measurements (which observe the time distance between pairs of events) can be made. Their results are stored in a time variable and can be used for succeeding time constraints. For greatest flexibility, MSC makes no assumption on whether the time domain is dense or discrete but leaves this decision to the developer.

Instance Creation and Stop. MSC provides constructs for creating and terminating instances. An instance is created by another instance. Graphically, this is expressed by a dashed arrow likewise a message. On the other, the termination is always caused by the instance itself. Due to the fact, that each newly created instance is shown explictly in an MSC, there is no way to express that one instance creates a varying number of child instances. Though this can be considered a restriction of the expressiveness of MSC, one has to keep in mind that MSC is not designed for describing complete systems but single scenarios where the number of instances involved is known.

5.1.2 Data Model

MSC does not have its own data language. Instead, it defines a general interface that allows to use type declarations, variables, and expressions with the syntax and semantics of arbitrary languages such as C, Java, SDL, or TTCN. This approach makes it possible to adopt the data model of the later implementation language and to use it already in the requirement specification phase. For instance, the notation and semantics of the SDL data model can be used for the specification of actions, conditions, and message/timer/instance creation parameters. Only very few requirements have to be met by the embedded data language. For instance, a boolean data type must be available for the definition of guarding conditions. Natural number expressions are required for specifying boundaries in loop expressions. Finally, a data type must be provided that is suitable for specifying time constraints.

5.1.3 Structural Concepts

Inline Expressions. Inline expressions allow to compose complex and alternative traces based on partial traces within a single MSC diagram. An inline expression consists of an operator and one or more MSC regions (i.e., events among a set of instances) to which the operator applies. MSC provides several inline operators to describe alternatives, parallel composition and loops: The `alt` operator is applied to two or more MSC sections. Its semantics is that exactly one of the alternatives is executed. The `opt` operator is used to describe optional events inside an MSC. Exceptional cases are handled best by the `exc` operator – either the events inside the inline expression are executed and the MSC is finished, or the events following the inline expression are executed. Both `opt` and `exc` expressions are syntactic shorthands and can easily be replaced by semantically

equivalent `alt` expressions. The `par` operator defines the parallel execution of MSC sections. No assumption is made about the overall event order except that the event order within each section is preserved. The `loop<`*min*`,`*max*`>` operator means that the events in the MSC section are executed *min* to *max* times. Instead of concrete upper and lower boundaries, the keyword `inf` can be used to specify infinite loops.

In figure 5.1(b), two nested inline expressions are shown. The loop expression means that message *MDATreq* is sent and received four times. The outer `alt` expression describes two different scenarios of the Inres protocol which are separated by a dashed line. The first alternative describes the case where a data transfer fails due to a problem with the unreliable medium and the *Initiator* sends a disconnection indication to *ISAP1*. In the second alternative, the data transfer is successful and the connection is terminated intentionally by *ISAP1* afterwards.

MSC References. In order to re-use MSCs in other MSC diagrams, MSC references can be used. A simple MSC reference consists of the name of the MSCs to be included and an optional list of actual parameters. More complex MSC references can be constructed by textual reference expressions that make use of the same operators as inline expressions plus an additional sequence operator `seq`. MSC references are interpreted rather in a macro-like than in a function-like manner, i.e., the execution of MSC instances is not synchronized implicitly before and after an MSC reference. MSC references result in a hierarchy of nested MSC diagrams if an MSC has a reference to another MSC that again includes an MSC reference.

In figure 5.1(b), a simple reference is made to MSC *ConnectionEstablishment* which is shown in figure 5.1(a). In figure 5.2 (see section 5.1.4 for a description of this diagram type), "*loop<0,inf> ConnectionFailure*" denotes an MSC reference expression which causes the execution of the events in MSC *ConnectionFailure* arbitrarily often (or never at all).

Decomposition. While MSC references allow to divide the message flow into different diagrams (by horizontal splitting), *instance decomposition* is a technique to describe the inner structure and behavior of a single instance kind by another MSC diagram (vertical splitting). Decomposition can be applied to describe the interaction between instances on different levels of detail. For example, in figure 5.1(a) *Initiator* is a block instance which can be subdivided into processes. Its internal structure is defined by an MSC called *IniProcs* (not shown here) as indicated in the instance header.

5.1.4 High-Level Message Sequence Charts

A *High-level Message Sequence Chart* (HMSC) describes how single MSCs can be combined. An HMSC consists of one or more directed graphs where each node is either a (unique) start node, an end node, a connector, an MSC reference (expression), a condition, or an HMSC itself. HMSCs do not consider single instances. Hence a condition in

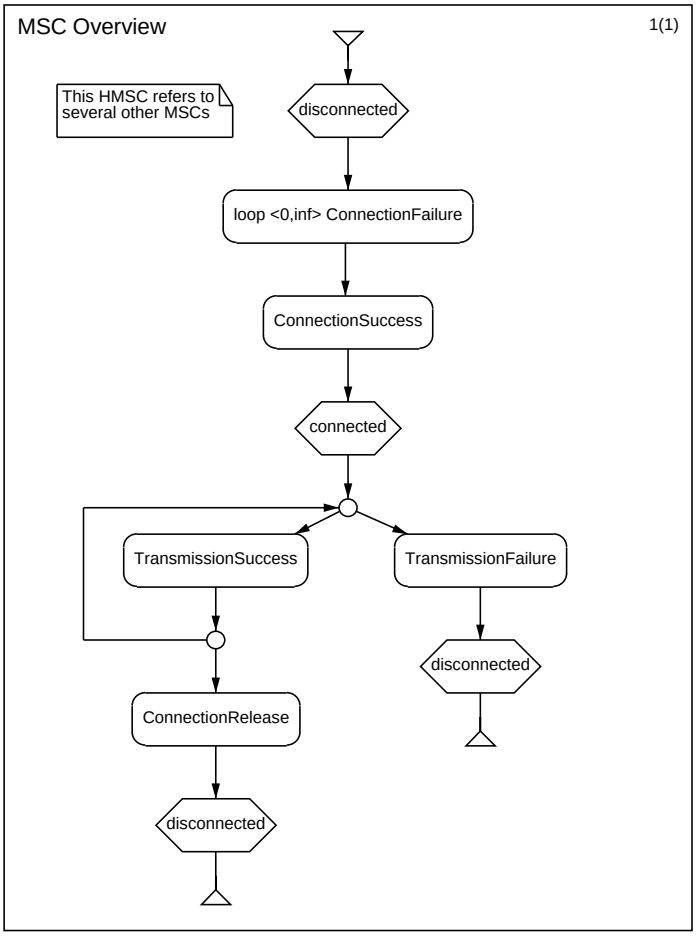


Figure 5.2: A High-level Message Sequence Chart

an HMSC refers to some global system state. Reference expressions used in an HMSC may either involve Basic MSCs and/or other HMSCs.

In figure 5.2, an HMSC is shown for Inres. After zero or more (potentially infinite many) failures, a connection is finally established. If the succeeding data transmission is successful, it can either be repeated or the existing connection can be released. In case any data transmission attempt fails, the system is directly set back into state *disconnected*.

Figure 5.2 illustrates two important aspects that hold for inline and reference expressions as well: First, an MSC does not necessarily describe deterministic behavior. In the given example, no assumption is made about when to leave the infinite loop covering the reference to *TransmissionSuccess*. The diagram only states that the connection may *eventually* be released. Second, alternatives are not required to be immediately distinguishable. For example, MSC *TransmissionSuccess* and *TransmissionFailure* may certainly start with the same messages. Therefore, MSCs are defined semantically by a *delayed choice* operator.

5.2 Specification and Description Language

The *Specification and Description Language* (*SDL*; see ITU-T 1999a) is a graphical language for the modeling of distributed systems. It allows to specify both the functional behavior, data, and the structure of a system.

SDL is a formal language with a non-ambiguous semantics based on abstract state machines (Prinz et al., 2000). Therefore, SDL specifications cannot only be used as informal addenda but also as normative parts in standardization. In addition, most SDL constructs can be transformed into efficient executable code which makes SDL also suitable as a high-level implementation language.

Conceptionally, an SDL specification can be considered a set of communicating extended finite state machines (CEFSMs) that are executed in parallel. Each CEFSM has its own local variables and timers. Communication among CEFSMs takes place by asynchronous message exchange over channels that connect sender and receiver. In addition, SDL provides syntactic shorthand notations for describing synchronous communication in the form of remote procedure calls which are modeled implicitly by the exchange of two signals.

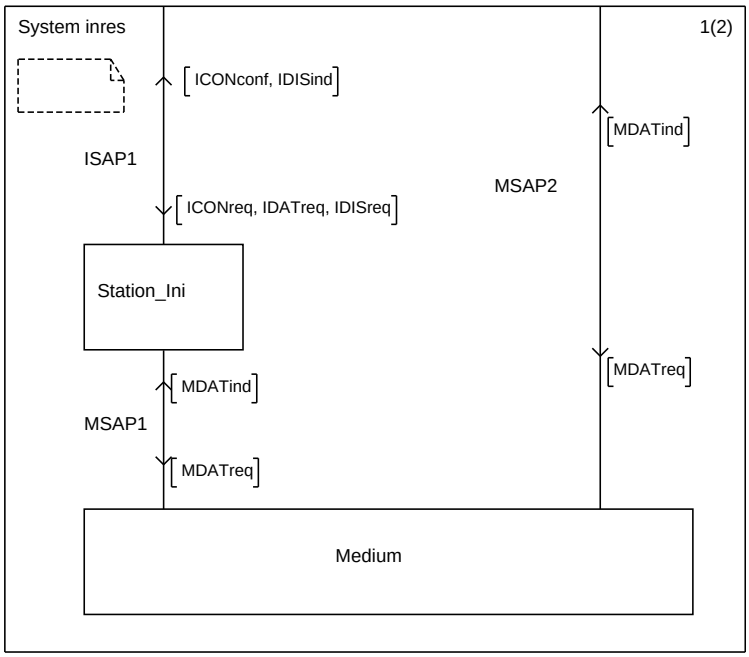
5.2.1 Agents and Structuring

An SDL specification consists of a number of diagrams which, in combination, describe the hierarchical structure of a distributed system. The building blocks of SDL are *agents*. An agent is described by one or more diagrams and may consist of a state machine, procedures, and variables. Furthermore, agents are allowed to contain other agents. There are two kinds of agents – *blocks* and *processes* – that differ in the degree of concurrency: In blocks, the state machine of the agent itself and the state machines of its embedded agents execute in *parallel*. In processes, the state machines are executed in an alternating manner. Transitions are interpreted sequentially and atomically, even if they involve several actions. The top-level block is called *system*.

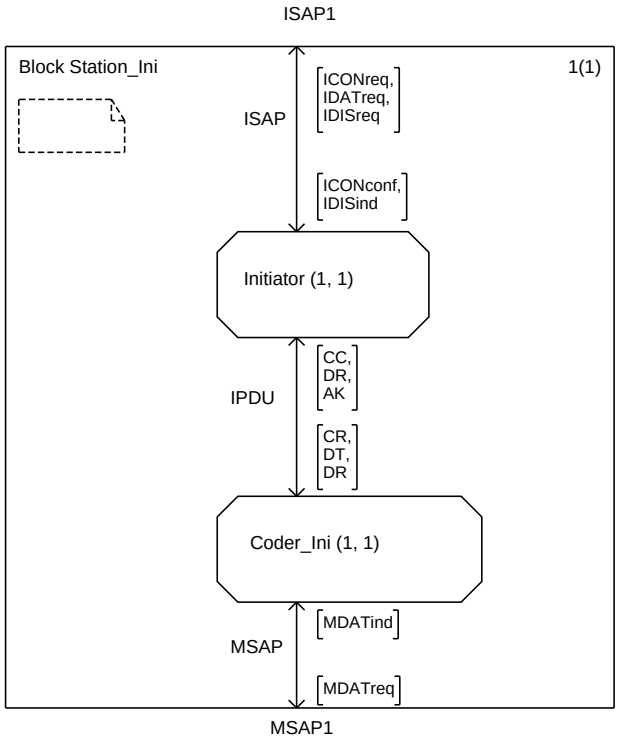
Figure 5.3(a) shows the SDL system diagram for Inres. The system consists of two blocks that represent logically independent entities. Block *Station_Ini* models a protocol instance at the sender side, block *Medium* describes the behavior of the underlying medium.

In a complete SDL specification, each agent that is referred to must be defined in a further diagram. A definition of block *Station_Ini* is given in figure 5.3(b). Once again, the block itself is composed of two process agents, named *Initiator* and *Coder_Ini*.

In principle, it is possible to associate state machines with blocks or to divide processes into blocks. However, traditionally, blocks are used to describe the hierarchical structure of a system according to some logical or implementation aspects, whereas processes are used on the lowest level for specifying the functional behavior.



(a) System specification



(b) Block *Station_Ini*

Figure 5.3: Inres – Structural description

5.2.2 Communication

In SDL, communication among the state machines of agents takes places by asynchronous exchange of signals. A signal is characterized by a name (which is the signal *type*) and an optional number of parameters.

In order to exchange signals, the sending and the receiving agent must be connected by a *channel*. Channels can be either unidirectional or bidirectional where the latter are modeled by two unidirectional channels. Furthermore, a channel may transmit signals either with or without delay.[3]

In figure 5.3(a), the two blocks *Station_Ini* and *Medium* are connected with each other by (delaying) channel *MSAP1*. In the SDL specification, Inres is modeled as an open system that interacts with its environment. The *Service Access Points* (*SAPs*) of the initiator and the medium (at the responder side) are associated with two channels called *ISAP1* and *MSAP2*.

For each direction of a channel, a list of valid signals has to be specified. For example, signals *ICONreq*, *IDATreq,* and *IDISreq* are allowed to be sent to *Station_Ini* via *ISAP1*. In the opposite direction, the block may respond with *ICONconf* or *IDISind*.

If a block or process agent communicates with its environment, one or more *gates* have to be specified at the diagram boundaries. These gates must be defined both in the diagram that defines the agent as well as in all diagrams that refer to the agent. Gates are required to ensure the consistency between the various diagrams of an SDL specification. By means of gates, it can be determined which state machines can communicate with each other, even if the communication paths are given indirectly by a hierarchy of diagrams. In figure 5.3(b), there are two gates named *ISAP1* and *MSAP1* which allow to embed the block diagram unambiguously into the system diagram in figure 5.3(a).

5.2.3 Behavior

The dynamic behavior of an SDL system is described by extended finite state machines. An EFSM is characterized by a finite number of states and state transitions. In SDL, each EFSM has one input queue in which all incoming signals are stored.

A state machine either waits for a new signal to arrive in its input queue or performs a state transition. The state machine and its input queue work independently such that incoming signals are not get lost, even if the EFSM performs a state transition. Two signals that arrive at the same time from different sources or via different channels are queued in arbitrary order. Since the exact execution order is not predictable in a distributed system, race conditions may occur that make the system behave indeterministically.

During a transition, a state machine can execute various actions, e.g., assigning a new value to a local variable. Typically, a state machine reacts upon an incoming signal

[3] Delay-free transmission does not imply synchronous communication as the receiving agent may not necessarily consume the incoming signal at the time of its arrival.

by sending one or more signals where the receiving agent is either stated explicitly or determined indirectly by the signal and channel involved.

Figure 5.4 shows an extract of the state machine for process *Initiator*. Coming from the start state (denoted by an empty oval), the state machine enters state *disconnected* first. Then it waits for the arrival of either signal *ICONreq* or *DR*. If, e.g., a connection is requested, process *Initiator* sets its local variable *counter* to *1*, sends signal *CR*, starts a timer (see below), and enters state *wait*.

For some problems it is inadequate that a transition is triggered by the first signal in the input queue. Therefore, a few additional language construct have been introduced into SDL that break the FIFO concept. If, at run-time, there is a signal in the input queue for which no transition is defined in the current state, the signal is discarded by default. In some situations it is desirable to keep such a signal until it can be evaluated. In process *Initiator*, a new request for data transfer (signal *IDATreq*) cannot be handled in state *sending* since the current transfer has to be completed first. The *save* construct (represented by a rhombus symbol) preserves the signal such that it can be processed later. Signals can also be prioritized with regard to a specific state. Finally, the consumption of signals can depend on a (Boolean) guard expression.

Timers. The behavior of an SDL state machine cannot only be triggered by incoming signals but also by timer expiration. Each state machine may have an arbitrary number of local timers on which the operations *set* and *reset* can be applied. Operation *set* activates a timer and has the expiration time as its parameter (typically, an expression such as `now+5`). *Reset* stops a timer. If a timeout occurs, a signal with the name of the timer is placed into the input queue.[4]

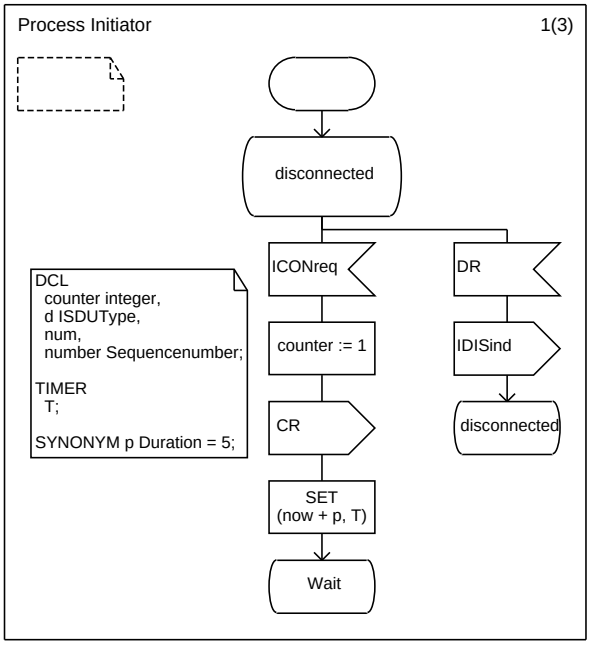
Timers are useful in order to prevent agents that wait for a particular input from being blocked. If an expected signal does not arrive within a given period of time, a supervision timer expires and a timer signal triggers a state transition. Timers can also be applied for describing events that re-occur with a certain time-lag.

In state *sending* of process *Initiator* (see figure 5.4(b)), timer T is used to guard the data transfer. If the receiving party does not acknowledge within a given period of time, T expires and the data is retransmitted. If signal AK is not received even after the fourth attempt, signal *IDISind* is sent in order to indicate that the connection has been closed.

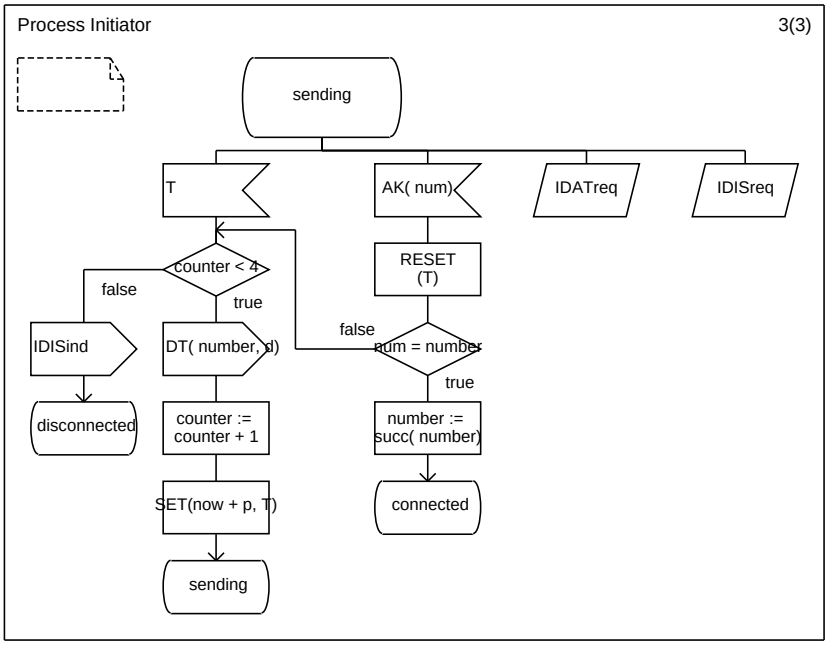
SDL does not define time units. Therefore it is left to the reader whether p in the text box of figure 5.4(a) is interpreted as 5 seconds, 5 light years, or another duration. In addition, no assumptions are made about the execution time of actions. For that reason it is possible that timers with the same duration expire in reverse order of their activation.

Dynamic Agent Creation. SDL supports the dynamic creation of agents. Therefore it is possible to describe, e.g., server architectures where each service request is handled

[4]Formally, a timer is modeled as a separate process that exchanges signals with its parent process.



(a) State *disconnected*



(b) State *sending*

Figure 5.4: Excerpt from the description of process *Initiator*

by a distinct agent. Each agent has a unique *process identifier* (pid). At agent creation, pids are exchanged implicitly between parent and child agent.

In SDL, no agent can enforce the termination of another one. Instead each agent is responsible for its own termination. However, it is possible to specify an agent in such a way that it accepts a particular signal in all states that makes the agent terminate.

5.2.4 Object Orientation

SDL fully conforms to the object-oriented programming paradigm. Just like data types, agent types, procedures, and signals can be specialized by inheritance. In this way it is possible to add new properties or to modify existing ones. With regard to agents this means that single state transitions can be overwritten in a derived agent.

All kinds of SDL types can be parameterized for easier re-use in different contexts (similar to the template concept in C++). Furthermore, type definitions can be combined in *packages* and imported by other SDL documents.

5.2.5 SDL Data Model and ASN.1

In contrast to MSC which allows to embed an external data language by some meta data concept, SDL provides its own data types and operations. Former versions of SDL were based on abstract data types whose properties were – among others – defined by axioms. Since these axioms cannot be evaluated efficiently by tools, the abstract data type concept has been dropped for SDL-2000 and replaced by a data model that has many similarities with the ones used in modern object-oriented programming languages.

SDL provides a couple of predefined standard data types such as *boolean*, *character*, *integer*, or *real* and a few special purpose data types that can be used for communication (*pids*) or timer operations (*duration*). Based on the simple data types, several complex data types can be constructed (*struct*, *array*, *choice*, etc.).

SDL distinguishes between values and objects. The latter are references to values. The semantics of SDL objects is very similar to the semantics of references in the *Java* programming language.

A specialty of SDL is the *any* expression which returns an unspecified value of a given data type. With *any* it is possible to describe nondeterminism in the behavior of a system.

In addition to SDL's own data language, the *Abstract Syntax Notation One* (*ASN.1*, see ITU-T 1997a) can be used. ASN.1 is a standardized notation for data types and values that is commonly used in the telecommunication area. Its main benefit is the definition of encoding rules that specify the representation of values on the bit level during transfer. Moreover, ASN.1 can be used in TTCN test suites which makes the data language an ideal mediator between SDL and TTCN. On the other hand, ASN.1 does not define any operations on its data types.

There are two standards that define how to use ASN.1 in combination with SDL. Z.105 (ITU-T 1999b) defines a mapping of ASN.1 constructs to equivalent SDL constructs and a small extension to SDL that allows to import data types and values from ASN.1 modules. Z.107 (ITU-T 1999c) goes even further by describing how ASN.1 notation can be used directly within an SDL specification.

5.2.6 Further Language Constructs

SDL provides a number of additional language constructs that cannot be discussed here in detail. For example, SDL has a number of control structures that allow to describe complex algorithms in both a graphical and a textual notation. Another feature of SDL is its exception handling that has been taken over from C++. For further information and a complete description of SDL, the reader is kindly referred to the SDL Forum Society.

5.3 Tool Support

The success of a language is closely coupled with the availability of powerful development tools. For MSC and SDL, there exist a number of commercial and academic software packages (SDL Forum Society, 2002b). Their tools range from graphical editors that support the developer with predefined symbols and incremental syntax checkers, to compilers for the automatic generation of C++ and Java code from SDL specifications.

The semantic closeness of SDL and MSC allows to use both languages in combination. For instance, during the step-wise simulation of an SDL specification, the current trace can be presented in form of an MSC. In the other direction, it can be verified whether the behavior of an SDL system conforms to some requirements expressed in an MSC.

In addition to simulators or debuggers, validation and verification tools can be applied that allow to explore the state space of an SDL system interactively or automatically and analyze the specification with regard to violations of the dynamic semantics of SDL (e.g., whether a signal is sent to a process which has already terminated). Such tools can also find other potential problems such as deadlocks or implicit signal consumptions.

A Syntax Checker for SDL-2000. Since its introduction in 1976, SDL has undergone regular modifications. These changes were motivated by new application areas, the need for object-oriented design, and harmonization with other languages such as C++ and UML. In November 1999, Study Group 10 of the ITU-T has approved the latest version of SDL, called SDL-2000. Because of its many changes (in particular in the organization of the standard and on the level of grammar rules), there is a strong demand for completely redesigned software tools.

The first step in the development chain of any SDL tool is the construction of a parser which builds an abstract syntax tree. However, even though SDL-2000 comes along with a complete grammar for the textual representation, it is not possible to simply

feed the grammar rules into a compiler construction tool. Instead, considerable effort has to be spent on transforming them into an appropriate input format and resolving nondeterminisms.

Within a project of 4 man months, the author has developed the first available SDL-2000 parser/syntax checker. Except for macros, it supports the complete SDL-2000 definition. Technically, it is based on the same concepts and tools as the TTCN-3 parser described in section 3.2.7 on page 40. The SDL parser has been used partially for the ValiBOSE project (section 10.3) as well as for external projects (e.g., Zieren, 2000) The work on the parser has unveiled several errors and inconsistencies in the Z.100 standard. Nine change requests have been submitted to the SDL rapporteur at ITU-T and taken into account for a *Master list of changes*. For detailed information on implementation issues, the reader is referred to Schmitt (2000).

6 The Autolink Tool

Within the AUTOLINK research and development project, a tool for the automatic generation of TTCN-2 test suites based on SDL system specifications and MSC test purposes has been developed. The AUTOLINK project was started in 1996 by the Institute for Telematics in Lübeck and TELELOGIC AB in Malmö, Sweden. It has been documented in a large number of publications, see, e.g., Ek et al. (1997); Grabowski et al. (1999); Koch et al. (1998); Schmitt et al. (1997, 1998, 2000); Schmitt and Koch (2001).

AUTOLINK is integrated into TELELOGIC's TAU development environment. TAU provides tools for the design, analysis, and compilation of systems and protocols specified in SDL, MSC, and TTCN. It supports most features of the SDL-96 standard and allows the combined use of SDL with ASN.1.

In practice, many test generation methods – such as the more advanced ones presented in section 4 – fail due to their complexity. Similarly, many academic test generation tools are practically useless due to implementation-specific restrictions. A major goal of the AUTOLINK project was to develop a tool that is capable to handle large-scale industrial specifications.

AUTOLINK has been used for in-house developments at all major telecommunication companies (NOKIA, ERICSSON, and MOTOROLA). Furthermore, standardized test suites have been developed at the European Telecommunications Standards Institute (ETSI).

During the five-year project, it has turned out that test specifiers are often tackling subtle problems that strongly differ from "classical" academic questions. Thus, the project was driven to a large extend by practical experience and user feedback.

In this chapter, the AUTOLINK test generation methodology is presented. In section 6.1, an overview of the AUTOLINK test generation process is given. The three phases of the AUTOLINK test generation process are described in sections 6.2, 6.3, and 6.4. Various ways of interpreting MSCs for test generation are discussed in section 6.5. Two AUTOLINK case studies are described in section 6.6. Finally, a comparison of AUTOLINK with other SDL-based test generation tools and a number of open issues are presented in sections 6.7 and 6.8.

Within the AUTOLINK project, solutions have been elaborated to generate test cases for distributed test architectures, to cope with the state space explosion problem, and to produce well-structured and readable test suites. These topics are described in detail in chapters 7 to 9.

6.1 The Autolink Test Generation Process

Test generation with AUTOLINK follows a three-step process that comprises the phases *test purpose specification*, *test case generation*, and *test suite production*. The overall process is outlined as a UML activity diagram in figure 6.1.

During test generation, a given SDL system is considered as the system under test and the SDL channels to the system environment are mapped to PCOs. Signals transmitted over these channels become send and receive test events in TTCN-2.[1]

In the first step, the test specifier has to define a set of test purposes. In AUTOLINK, a test purpose is a sequence of input and output events that are to be exchanged between the SDL system and its environment. Test purposes are developed either manually, interactively, or fully automatically. AUTOLINK uses MSC as a uniform format for the representation of test purposes.

Based on a set of MSC test purposes, test case generation takes place. As result, an internal data structure is constructed that describes the dynamic behavior of each test case and the associated constraints.[2] Normally, a test case is computed by a state space exploration of the synchronized product of the SDL system and an MSC. However, sometimes it is impossible to simulate/verify an MSC test purpose. In these cases, the MSC can be transformed directly into an internal test case representation.

In a final step, a TTCN-2 test suite in MP format is produced based on the internal representations. It comprises tables in the declarations part, constraints part, and dynamic part.

The test generation process is controlled by several user-defined options. Among others, they allow to customize the appearance of the generated test suite with regard to constraints, test configuration, and test suite structure.

6.2 Test Purpose Specification

AUTOLINK derives test cases from paths which have to be provided by the test specifier. A path is a sequence of SDL events which drive the system from a start state to an end state in the state space of the SDL system. A path is stored as a Message Sequence Chart. Normally, an MSC test purpose generated by AUTOLINK only shows the externally observable interaction that takes place between the SDL system and its environment. It consists of one instance axis representing the SDL system and one instance axis for each channel linked to the environment.

A typical MSC for the Core INAP CS-2 protocol (see section 6.6.1) is shown in figure 6.2. Instance *CS2_INAP* denotes the system and *SCF*, *SigCon_A*, and *SigCon_B* represent channels to the test environment that are mapped to PCOs in TTCN-2.

[1] In the following, if no source or destination is specified, a communication event is described from the point of view of the test environment. That means, a *send event* refers to a message sent from the test environment via some PCO/channel to the system, whereas a *receive event* is received by the tester.

[2] The term *constraint* is used according to the TTCN-2 terminology.

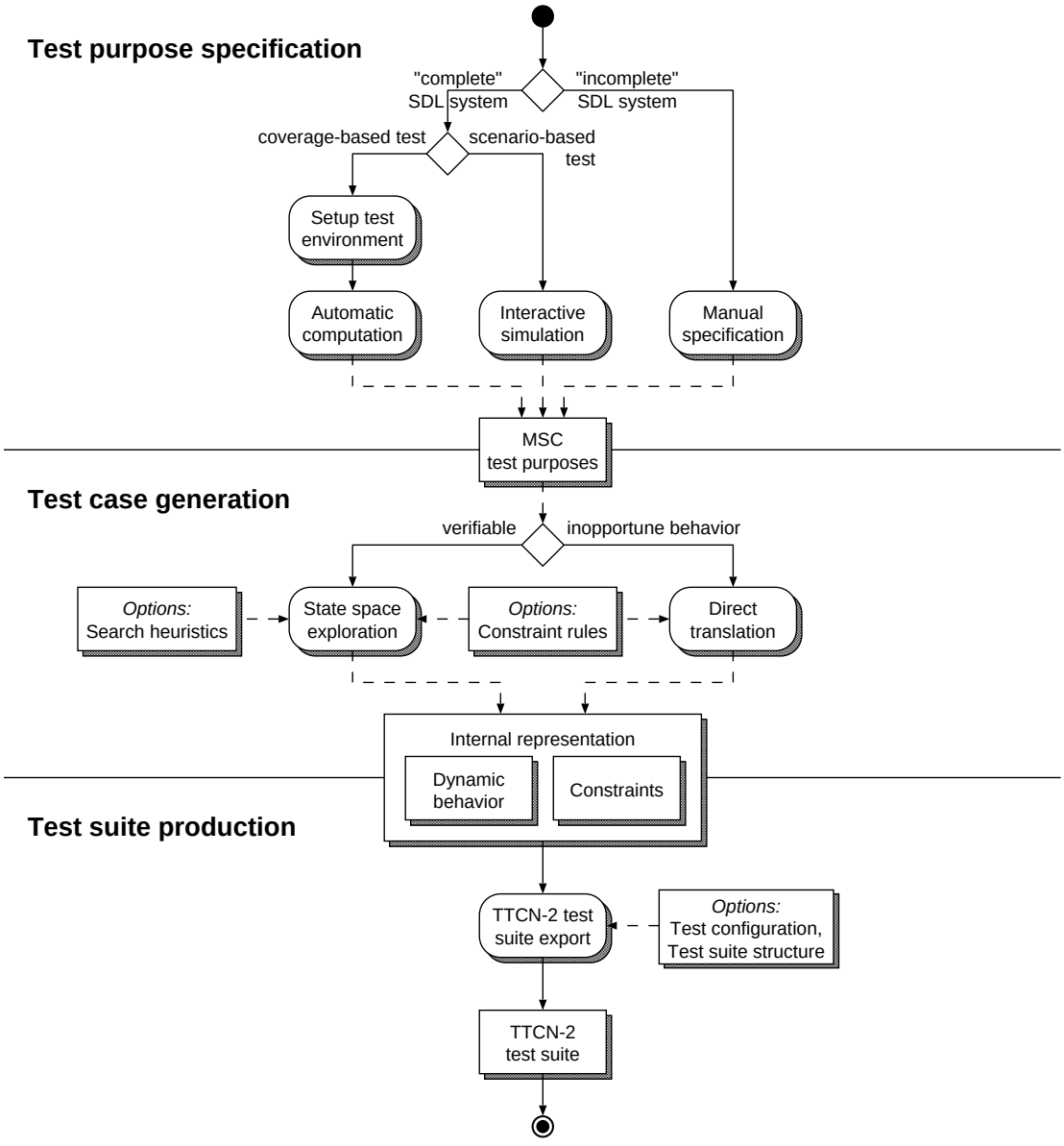


Figure 6.1: Test suite generation with AUTOLINK

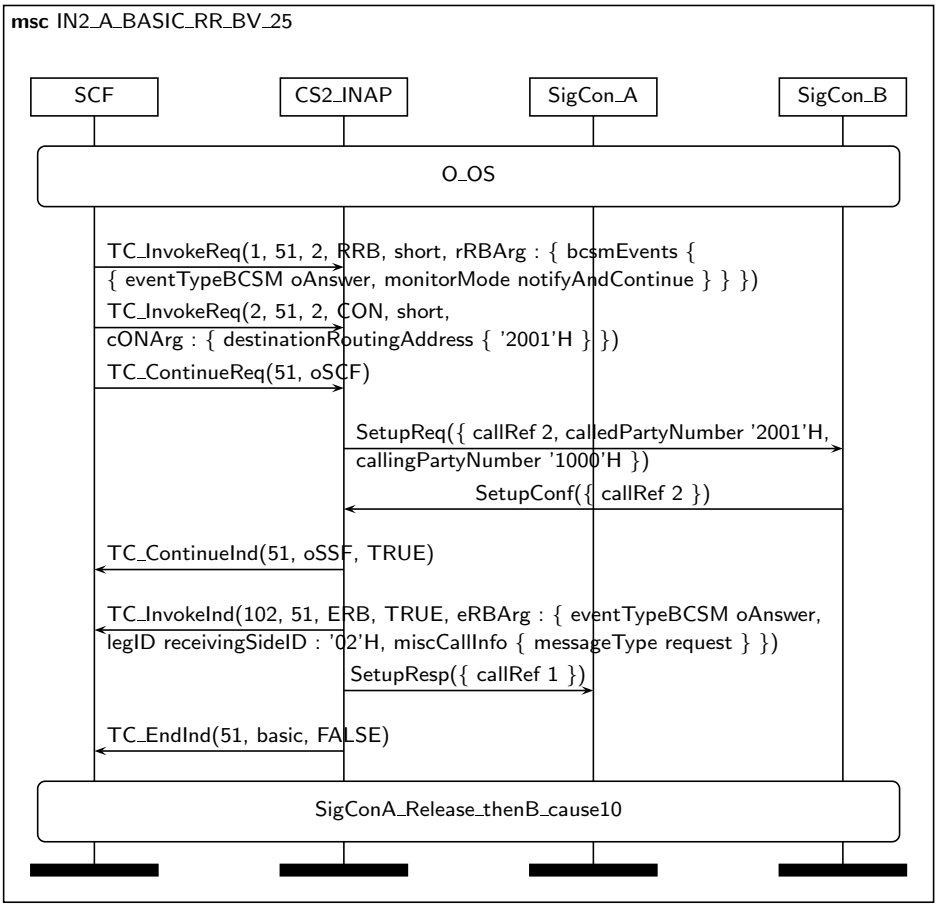


Figure 6.2: MSC *IN2m_A_BASIC_RR_BV_25*

Test purposes can be specified in several ways: manual specification, interactive simulation, observer processes, and automatic computation. The different approaches are described in the following.

6.2.1 Manual Specification

MSC test purposes can be specified manually by using an MSC editor. Manual specification should only be considered in cases where MSCs are specified prior to the SDL system or an MSC describes a scenario that is not described by the SDL system.

6.2.2 Interactive Simulation

In order to produce MSC test purposes by means of state space exploration, the test specifier has to define a set of reasonable input signals. Whenever the SDL system is in a stable state, i.e., in a state in which the system waits silently for new input from its

environment or the expiration of a timer, AUTOLINK continues the simulation with each possible input.

For complex specifications, it is difficult to predict a set of inputs *in advance* that results in a good coverage. In addition, due to the fact that each possible signal is tested in each stable state, the state space may grow excessively with increasing depth. For one of the test suites of Core INAP CS-2, the SDL system had to be controlled by more than 130 (!) different input signals.

On the other hand, the test specifier usually knows exactly which signal (parameters) make sense at which point in time. Therefore, he may want to specify test purposes by stepwise simulation. This approach corresponds to testing based on scenario-based requirements as described in section 4.2.3.

6.2.3 Observer Processes

An *observer process* is a special kind of SDL process that allows to check some property of an SDL system state. Observer processes are able to inspect the SDL system without interfering with it and to log the fulfillment of their property. To accomplish this, three features are implemented in the TAU VALIDATOR:[3]

- *The observer process mechanism*

  By defining processes to be observer processes, the TAU VALIDATOR simulates an SDL specification in a two-step manner. First, the rest of the SDL system executes one transition, and then all observer processes execute one transition and check the new system state.

- *The assertion mechanism*

  The assertion mechanism enables an observer process to generate reports during state space exploration. These reports include the complete search path and can be transformed into MSCs later.

- *The* Access *abstract data type*

  The *Access* abstract data type allows observer processes to examine the internal states of other processes in the system, i.e., it can check variable values, contents of queues, etc., without having to modify the observed processes.

The general idea when using observer processes for test generation is to describe one or more test purposes by an observer process. Then, a corresponding path that satisfies the requirement(s) expressed by the observer process is searched by space state exploration.

Since observer processes as implemented in the TAU VALIDATOR are able to access variables and functions of the VALIDATOR's run-time environment, they can also be

[3] Observer processes are non-standardized, tool-specific extensions. For instance, observer processes in OBJECTGEODE (called *goal observers*) have three kinds of states: *Accepting states* indicate that the executed transitions of system are valid. *Success states* indicate that the observed sequence of system events meets some requirement. *Error states* indicate an incorrect sequence of system events. Whenever an observer reaches a success or error state, the current simulation path is pruned.

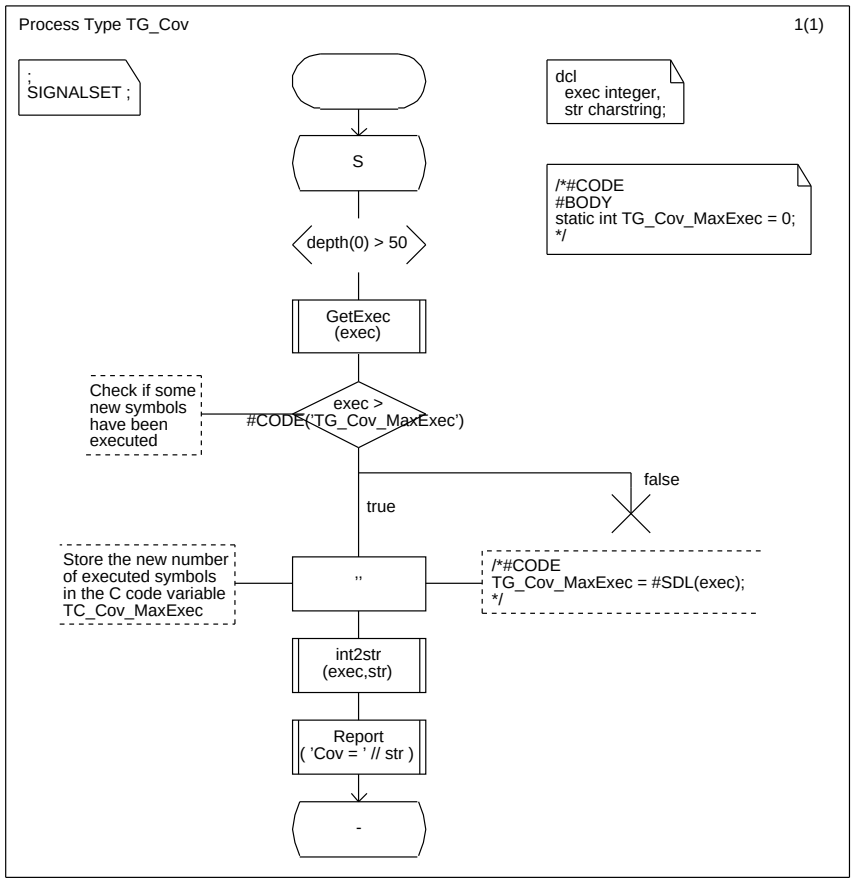


Figure 6.3: An observer process for test generation

used for controlling the state space exploration and encoding general test criteria. In this case, the application of an observer process may lead to several MSC test purpose.

A simple observer process type is shown in figure 6.3. *TG_Cov* realizes a coverage-based test method. Each time a trace with length 50 is executed that covers some additional SDL symbol, a new report is generated. *TG_Cov* is intended to be used in combination with the Random Walk search strategy that repetitively explores random paths through the state space (for a critical consideration of Random Walk see section 8.1 on page 120).

6.2.4 Automatic Computation

AUTOLINK allows to derive MSC test purposes fully automatically from an SDL specification. Based on a state space exploration, a set of test sequences is computed whose execution shall result in a large structural coverage of the SDL specification.

AUTOLINK defines coverage on the basis of single SDL symbols. However, the execution of a coverage unit may not be observable when it comes to black-box testing, because it does not necessarily involve interaction of the SDL system with its environment.

Therefore, generating a separate test purpose for each coverage unit leads to many identical test cases.

To circumvent the problem, AUTOLINK examines larger sequences of coverage units that lead from one stable state to another one. Each automatically generated test purpose covers at least one unique observation step. In most cases, an observation step includes a stimulus from the test environment and one or more corresponding responses from the system.

Unfortunately, there is no universally applicable search strategy to find paths through the reachability graph which results in a large coverage. General-purpose search strategies such as depth-first search, breadth-first search, or Random Walk lead to poor coverage or the computed test purposes include many redundant events. For that reason, AUTOLINK provides a new test generation algorithm, called TREE WALK, that is described in detail in chapter 8.

6.3 Test Case Generation

In the test case generation phase, an internal test case representation is generated for each MSC test purpose. The dynamic behavior description contains all sequences of test events that lead to either a pass or an inconclusive verdict. Send and receive events are associated with constraints codifying the SDL/MSC signal parameters. Since constraints can be shared among several events in different test cases, they are stored separately from the test case representations.

AUTOLINK merges identical constraints and resolves naming conflicts (for the issue of constraint naming see also chapter 9.1.2). Furthermore, a configuration language is provided that allows the test specifier to define rules for the mapping of SDL signals to TTCN-2 constraints (see chapter 9.1.2).

There are two approaches to generate test cases based on MSC test purposes that are explained in the following subsections.

6.3.1 State Space Exploration

Ideally, AUTOLINK generates a test case by parallel simulation of the SDL system specification and a given MSC test purpose (*synchronous product simulation*). During the simulation, the SDL system is only triggered by input signals that are specified in the MSC. In the opposite direction, additional valid signals (test events) can be determined which are not specified in the test purpose but which the SDL system (i.e., the SUT) is allowed to send to its environment (i.e., the tester). These test events result in an *inconclusive* test verdict.

During simulation, each state is a pair $\langle SDLState, MSCState \rangle$. Since AUTOLINK follows the black box testing approach, only communication between the SDL system and its

environment must be considered. Therefore, MSC instances that correspond to internal components of the SDL system are disregarded.

During simulation, for every event that occurs in a transition of the SDL system, one of the following cases must be considered:

1. The event belongs to the class of internal events ⇒ execution proceeds with the successor state of the SDL system and the current MSC state.
2. The event matches with an event specified in the MSC ⇒ the event is added to the test case tree as an event leading to a leaf node with *pass* verdict; the execution proceeds with the successor state of both the SDL system and the MSC.
3. The observable event conflicts with a different event in the MSC ⇒ the event is added to the test case tree as a leaf node with *inconclusive* verdict.

Like any other verification, validation, or test generation tool, AUTOLINK has to cope with the state space explosion problem, i.e., the reachability graph grows exponentially with increasing depth. Due to its complexity, it is impossible to store it completely in memory. Therefore, AUTOLINK constructs the reachability graph *on-the-fly*. While traversing, only the system states along the current path are stored. During backtracking, state information is removed from memory.

AUTOLINK performs its state space explorations based on the well-known *Supertrace* algorithm (Holzmann, 1991). It allows to identify states that have been visited before along a different path.

Furthermore, AUTOLINK supports several heuristics that avoid the analysis of system traces which are supposed to be irrelevant. Heuristics are based on assumptions and experience (rules of thumb). Grabowski et al. (1996) distinguish between different types of heuristics: *Limiting heuristics* restrict the length of traces. Possible limiting criteria are a fixed maximum number of events or a specific state property.[4] *Filtering heuristics* aim at reducing the branching factor at some states such that the behavior tree becomes thinner. This can be achieved, e.g., by assigning different priorities to classes of events, restricting unbounded queues, or defining a limited set of valid input/output signals. As as third category, the authors present *SDL-specific heuristics*. For instance, the degree of concurrency can be reduced by executing a complete SDL state transition as an atomic event.

In AUTOLINK, various limiting and SDL-specific heuristics can be used for automatic test purpose computation and test case generation: The queues of channels that connect two internal processes/blocks can be deactivated if these processes/blocks are supposed to communicate without delay in a concrete implementation; in each system state, only one process instance may be considered, even if several processes can perform some action; during test case generation, inputs from the environment have priority over internal events of the SDL system, because it is assumed that a tester is faster than its SUT.

[4] According to Grabowski et al. (1996), approaches that stop exploration if a state is revisited fall into this category as well. However, these approaches are no real heuristics because soundness can be proven.

Further heuristics concern the capacity of input queues of processes, the atomicity of SDL transitions, and the time it takes to execute an action (the latter is relevant to determine when a timer can expire).

6.3.2 Direct Translation

If an MSC test purpose covers aspects of a protocol specification which are not modeled in the corresponding SDL system, it is obviously not possible to generate a test case by state space exploration. However, for a uniform test suite development process, it is desirable to formalize *all* test purposes as MSCs.

Therefore, AUTOLINK allows to convert MSCs directly into TTCN-2 test cases. Although no state space exploration is performed, AUTOLINK requires some information about the interface of the specification to find out which MSC instances represent PCOs and to check the syntax of the MSC with regard to the signals and signal parameters. An SDL system has to be provided which at least defines the channels to the system environment, the signals sent via these channels, and – for the sake of well-formedness – a dummy process.

Direct translation must be applied with caution. There is no guarantee that an MSC describes a valid trace of the specification or the implementation, respectively. Furthermore, it is impossible to compute test events which lead to an *inconclusive* test verdict, i.e., any deviation from the behavior described in the MSC is considered as a failure.

On the other hand, there are good reasons to use MSCs instead of directly writing TTCN test cases. First, test cases typically span trees with several leaves because of the partial order of test events.[5] In MSC, the partial order is expressed implicitly. While it is arduous for a test specifier to write down a complete TTCN test case, AUTOLINK automatically computes all valid permutations of test events. Second, since AUTOLINK always translates MSCs into an intermediate internal test case representation, test cases generated by direct translation can be merged with test cases generated by state space exploration. This leads to uniform and compact test suites with, e.g., a reduced number of constraints.

6.4 Test Suite Production

In the test suite production phase, AUTOLINK creates a TTCN-2 test suite in MP format based on the SDL system and the internal test case representations. In figure 6.4, a TTCN-2 test case is shown that corresponds to the MSC test purpose in figure 6.2 on page 82.

[5] The only way to avoid multiple branches with identical continuations is to introduce a test step for any unordered sequence of events. However, since the *send-first* rule is applied — i.e., if the tester can send a message, it will do so immediately without considering incoming messages — it can become difficult to identify a test step. In TTCN-3, the situation has been improved by the *interleave* operator in combination with the sequential execution of test cases.

Test Case Dynamic Behaviour					
Test Case Name : IN2_A_BASIC_RR_BV_25					
Group :					
Purpose :					
Configuration :					
Default : OtherwiseFail					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		+O_OS_2			
2		SCF ! TC_InvokeReq	CIR_RequestReportBCSMEvent_028(1, 51)		
3		SCF ! TC_InvokeReq	CIR_Connect_001(2, 51)		
4		SCF ! TC_ContinueReq	C_TC_ContinueReq_001(51)		
5		SigCon_B ? SetupReq	C_SetupReq({ callRef 2, calledPartyNumber '2001'H, callingPartyNumber '1000'H })		
6		SigCon_B ! SetupConf	C_SetupConf({ callRef 2 })		
7		SCF ? TC_ContinueInd	C_TC_ContinueInd_001(51)		
8		SCF ? TC_InvokeInd	CII_EventReportBCSM_012(102 , 51)		
9		SCF ? TC_EndInd	C_TC_EndIndBasic_003(51)		
10		SigCon_A ? SetupResp	C_SetupResp({ callRef 1 })	(PASS)	
11		+SigConA_Release_thenB_cause10_2			
12		SCF ? TC_AbortInd	C_TC_AbortInd(51)	INCONC	
13		SigCon_A ? SetupResp	C_SetupResp({ callRef 1 })		
14		SCF ? TC_EndInd	C_TC_EndIndBasic_003(51)	(PASS)	
15		+SigConA_Release_thenB_cause10_2			
16		SigCon_A ? SetupResp	C_SetupResp({ callRef 1 })		
17		SCF ? TC_InvokeInd	CII_EventReportBCSM_012(102 , 51)		
18		SCF ? TC_EndInd	C_TC_EndIndBasic_003(51)	(PASS)	
19		+SigConA_Release_thenB_cause10_2			
20		SigCon_A ? SetupResp	C_SetupResp({ callRef 1 })		
21		SCF ? TC_ContinueInd	C_TC_ContinueInd_001(51)		
22		SCF ? TC_InvokeInd	CII_EventReportBCSM_012(102 , 51)		
23		SCF ? TC_EndInd	C_TC_EndIndBasic_003(51)	(PASS)	
24		+SigConA_Release_thenB_cause10_2			
25		SCF ? TC_AbortInd	C_TC_AbortInd(51)	INCONC	
26		SigCon_B ? SetupReq	C_SetupReq({ callRef 2, calledPartyNumber '2000'H, callingPartyNumber '1000'H })	INCONC	
Detailed Comments :					

Figure 6.4: TTCN-2 test case *IN_A_BASIC_RR_BV_25*

SDL sort definitions are mapped to ASN.1 type definitions in the TTCN-2 declarations part. ASN.1 data types defined externally in an ASN.1 module are listed as *ASN.1 type definitions by reference* in TTCN. SDL signal definitions become ASN.1 ASP or PDU type definitions.

The appearance of the test suite can be controlled by various options. For example, constraints can be stored either as ASN.1 PDU or as ASN.1 ASP constraints. Test steps can be stored globally in the test step library, as local trees attached to a test case, or printed inline. In addition, the test specifier can decide whether a monolithic tester or a distributed test architecture is required.

When exporting a test suite, AUTOLINK checks the consistency of the test cases. For example, an MSC which is used as postamble for more than one test case does not necessarily result in identical test steps. Thus, if these test steps are stored globally, AUTOLINK has to assign unique names to them (by adding a sequence number). Furthermore, naming conflicts are resolved for parameters which are used with varying values in several test cases.

AUTOLINK assumes *trace preorder* as conformance relation. That means, the implementation must show only a subset of the behavior of the SDL specification. This is achieved by introducing a default behavior table into the TTCN-2 test suite that makes test execution fail for all events that are not defined by the specification.

6.5 Interpretation of MSC Test Purposes

There are different possibilities to interpret MSCs for test generation purposes. In the following, a few aspects of a mapping to TTCN-2 test cases are discussed.

6.5.1 Partial Order Semantics

In MSCs, events of different instances are only partially ordered, i.e., typically more than one valid sequence of events exists. If a test generation tool would generate all possible test sequences, then send events could appear as alternatives to receive events in TTCN test cases, rendering them indeterministic and invalid.

For transforming an MSC into a TTCN-2 test case, signals sent from the tester environment to the system have a higher priority than signals in the opposite direction by default. This is motivated by the fact that the tester is supposed to be faster than the SUT. Moreover, it avoids potential deadlocks during test execution if a test case is generated by direct translation. If there are several possible send events, an arbitrary order is defined for them.

However, there are situations where this strategy leads to incorrect test cases. Therefore, AUTOLINK allows to introduce *synchronization points* by means of MSC conditions that block the evaluation of events until all events above the synchronization point have been executed. In chapter 7 on page 103, a formalism is introduced that generalizes the usage

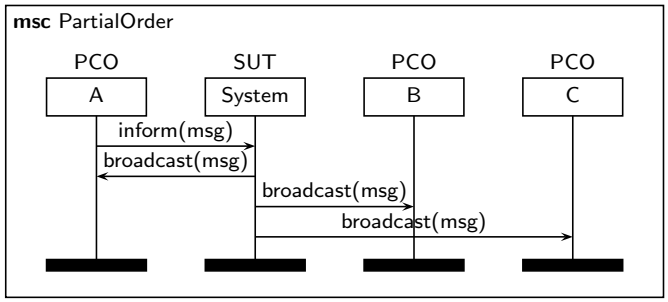


Figure 6.5: Message Sequence Chart *PartialOrder*

of MSC conditions for the synchronization of PCOs that are mapped to distributed test architectures.

When it comes to unordered events received from the system, there are two possible approaches to handle them:

1. Create a behavior description that is structured as a tree with branches wherever the order of events is not fixed.
2. Define some (arbitrary) order of events and create a behavior description with only one test sequence resulting in a pass verdict.

The difference between these approaches is illustrated by two test case descriptions in figure 6.6 that are derived from the MSC in figure 6.5. A test case description with only one valid path can be significantly shorter than a tree-like representation. On the other hand, error diagnosis might be more difficult with a linear sequence, since the test case might be stopped at an early stage due to an unreceived event while there are still unprocessed messages at other PCOs. Moreover, the specification of failures is more complicated as it is no longer valid to add a statement to the default behavior table that assigns a FAIL verdict to all unexpected events. AUTOLINK supports both approaches because either has its pros and cons.

6.5.2 Structuring Concepts

MSC References. In general, test cases are structured logically into several test steps, for example a *preamble*, a *test body* and a *postamble*. The distinction between several logical parts of a test case can be expressed in an MSC test purpose by using MSC references. Figure 6.2 on page 82 shows an MSC with two references. Preamble *O_OS* drives the SUT into the testing state and postamble *SigConA_Release_thenB_cause10* drives it back into the initial state. Test steps may refer to other test steps. During test case generation, AUTOLINK keeps track of the nested structure of test cases and test steps.

For the purpose of test generation, the semantics of MSC references (and reference expressions) has to be modified. ITU-T recommendation Z.120 considers references as macros, i.e., they are replaced with the content of the referred MSC for semantic analysis.

Test Case Dynamic Behaviour					
Test Case Name : PartialOrder					
Group :					
Purpose :					
Configuration : StandardConfiguration					
Default : OtherwiseFail					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		A ! inform	c_inform		
2		A ? broadcast	c_broadcast		
3		B ? broadcast	c_broadcast		
4		C ? broadcast	c_broadcast	PASS	
5		C ? broadcast	c_broadcast		
6		B ? broadcast	c_broadcast	PASS	
7		B ? broadcast	c_broadcast		
8		A ? broadcast	c_broadcast		
9		C ? broadcast	c_broadcast	PASS	
10		C ? broadcast	c_broadcast		
11		A ? broadcast	c_broadcast	PASS	
12		C ? broadcast	c_broadcast		
13		A ? broadcast	c_broadcast		
14		B ? broadcast	c_broadcast	PASS	
15		B ? broadcast	c_broadcast		
16		A ? broadcast	c_broadcast	PASS	
Detailed Comments :					

(a) TTCN-2 test case description with multiple paths

Test Case Dynamic Behaviour					
Test Case Name : PartialOrder					
Group :					
Purpose :					
Configuration : StandardConfiguration					
Default :					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		A ! inform	c_inform		
2		A ? broadcast	c_broadcast		
3		B ? broadcast	c_broadcast		
4		C ? broadcast	c_broadcast	PASS	
5		A ? OTHERWISE		FAIL	
6		B ? OTHERWISE		FAIL	
7		A ? OTHERWISE		FAIL	
Detailed Comments :					

(b) TTCN-2 test case description with only one valid test sequence

Figure 6.6: Representation of partially ordered events in TTCN-2

Since there is only a partial ordering among different instances, MSC permits traces where the order of events does not reflect the order of the MSCs involved. For example, in figure 6.2 the first *TC_InvokeReq* signal might be sent before MSC *O_OS* is evaluated completely. In order to be able to preserve the MSC structure, AUTOLINK requires that a test step is evaluated as a unit. Therefore, an implicit synchronization is made before and after each reference.

Inline and Reference Expressions. A system under test may not behave deterministically. By the use of *inline expressions* and *reference expressions*, it is possible to describe test cases where the tester reacts flexibly depending on the system behavior.

Moreover, if some test cases differ only slightly, inline and reference expressions can be used to describe different behavior of the tester. In that case, separate test cases are generated.

The following operators are supported in MSC expressions:

- The *alternative operator* (*alt*) is suitable for the description of situations where the continuation of a test case depends on the former output of the system. If both alternatives start with a message sent from the system to the test environment, two branches are generated within a single test case. The alternative operator may also be used to specify two alternative test sequences. If both alternatives start with a signal sent by the test environment, two distinct test cases are generated.
- The *optional operator* (*opt*) can be used, e.g., to specify messages which may or may not be sent by the system or to react to unexpected signals in a way that the test case can be continued normally afterwards.
- The *exception operator* (*exc*) is intended to be used for error handling. An exception expression may contain signals which prevent the test system from continuing the regular test execution. Optionally, an exception includes a sequence of signals which drives the SUT back into a stable testing state. An exception always results in an INCONC verdict.
- The *loop operator* (*loop*) can be used to describe the iterative execution of a (portion of a) test case.
- Finally, the *sequence operator* (seq) can be used within reference expressions in order to state that one test step follows another.

Of course, the usage of the different operators is not restricted to the applications described above. On the other hand, not all MSCs containing inline or reference expressions describe reasonable test cases. E.g., if one alternative starts with a send event from the tester while the other starts with a sent event from the system (considering of the preference of send events), the MSC describes an invalid test case.

As mentioned above, an implicit synchronization is made at the beginning and the end of MSC references. MSC reference expressions are a generalization of plain MSC references. For that reason, it makes sense to synchronize at MSC references, too, and to generate distinct TTCN test steps for each of the MSCs involved in the expression.

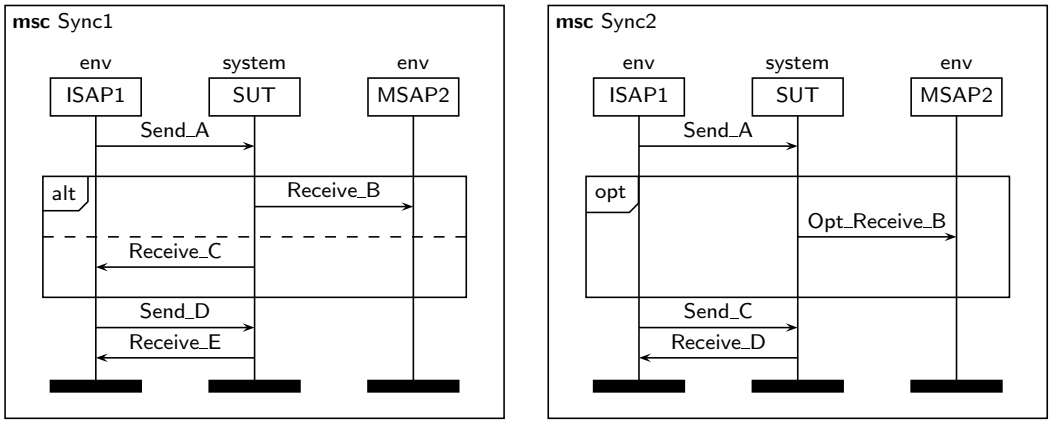


Figure 6.7: Synchronization among inline expressions

When it comes to inline expressions, it is not necessary to synchronize since they do not result in different test steps. Nevertheless, it has been decided to do so for consistency reasons. There are cases where synchronization among inline expressions is preferable, while in other situations, the resulting test case does not match the intention of the test specifier or even no valid test case can be produced.

In figure 6.7, two examples are given. With synchronization at the inline expression, MSC *Sync1* maps to a test case that waits for *ReceiveB* or *ReceiveC* before test execution proceeds with *Send_D*. If AUTOLINK did not synchronize at the end of the alternative expression, *SendD* would be sent before *ReceiveB* (please note that send events are prioritized). Moreover, since both alternatives are evaluated independently and combined in a single behavior tree afterwards, the send event *env1 ! Send_D* would become an alternative to the receive event *env1 ? ReceiveC* . This, of course, is not allowed in TTCN-2.

For MSC *Sync2* in figure 6.7, the following faulty event tree is generated in case of synchronization:

```
Env1 ! SendA
  Env2 ? OptReceiveB
    Env1 ! SendC
      Env2 ? ReceiveD
  Env1 ! SendC
    Env2 ? ReceiveD
```

Whenever there is a conflict between a send and a receive event, the test case is erroneous.

High-Level MSCs. HMSC diagrams can be used to illustrate the relationship between various test cases. For example, even though test cases normally have different test purposes, they might share the same preamble and postamble. This commonness can be expressed graphically by the use of an HMSC diagram such as the one in figure 6.8.

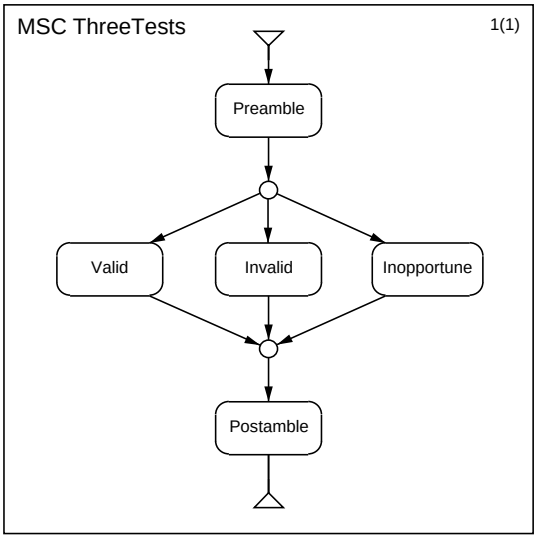


Figure 6.8: Three test cases described by one HMSC diagram

AUTOLINK generates separate test cases for each possible path through an HMSC. Of course, HMSCs may have more than one node with several outgoing edges, resulting in a potentially large number of test cases. In addition, each path itself may include HMSCs or MSC reference expressions and hence describe several test cases.

All test cases must have unique names. Thus, whenever there is a branch in the HMSC, the names of the succeeding MSC references are postfixed to the name of the top-level MSC (separated by '_'). With regard to HMSC *ThreeTests* in figure 6.8, the resulting test cases will be named *ThreeTests_Valid*, *ThreeTests_Invalid*, and *ThreeTests_Inopportune*.

6.6 Case Studies

The usefulness of AUTOLINK has been demonstrated in several case studies at the Institute for Telematics, TELELOGIC, BOSCH TELECOM, and NOKIA (see, e.g., Dai, 1999; Koch, 2001, chapters 8 and 9; Mayer, 2000). In the following, two case studies are described in which comprehensive standardized test suites have been developed at ETSI. They will also be used for illustration purposes in the following chapters in which particular aspects of test generation are considered.

6.6.1 Core INAP CS-2

The Core Intelligent Network Application Protocol (Core INAP) was the first protocol specified by ETSI for which a machine-processable SDL model is available (ETSI, 1999b). The SDL model was developed by ETSI Sub-Technical Committee SPS3 with support by the Protocol Expert Group and the Technical Committee 'Methods for Testing and Specification'.

The specification of INAP Capability Set 2 (CS–2) makes use of the object-oriented features of SDL-96 by inheriting CS–1. Data types are defined in ASN.1.

The SDL specification of ETSI's INAP CS–2 is voluminous. It comprises more than 450 pages in printed form. The SDL phrase representation is about 1.6 MByte large (approximately 570 KByte without comments). When translating the specification into C with TAU's code generator, about 350,000 lines or 13.6 MByte of source code are generated.

Test Suite Generation. Three TTCN test suites for INAP CS–2 were developed by ETSI Specialists Task Force STF 100. A test suite which covers the basic capability set, i.e., the CS–1 operations with CS–2 additions, was published as the first subpart of ETSI (2000b).

With respect to the CS–1 operations, test purposes were defined with textual descriptions and rough MSCs, first. Next, these test purposes were formalized as detailed MSCs using the TAU SDL SIMULATOR. In total, STF 100 specified 126 test purposes (ETSI, 2000a). For 67 test purposes, the MSCs could be simulated in order to produce the corresponding test cases. The remaining 59 test purposes had to be translated directly into TTCN due to unspecified parts in the SDL model.

The test suite resulted from a repetitive process of SDL/MSC refinements and modifications, MSC verifications, and test generation runs. Whenever a modification of the SDL model was made, all MSCs were verified with the TAU VALIDATOR. If an MSC could not be verified, the SDL model or the MSC were modified again until all MSCs passed the verification. Thereafter, the test case generation was started using AUTOLINK.

Statistics. Both the MSC verification and the test generation runs were executed at the Institute for Telematics in Lübeck. The test results discussed below were obtained on SUN ULTRA 2 workstations with 300 MHz processors.

Figure 6.9 shows the computation time of both the MSC verification and the test generation with AUTOLINK. The time needed for the verification of an MSC ranged from 1 min 24 sec to 2 h 15 min. It took between 6 min 44 sec and 51 h 49 min (= 3109 min) to generate a test case.

The larger amount of time needed for test generation is not surprising: During MSC verification, a path in the state space graph is truncated as soon as an event in an SDL transition conflicts with the MSC. On the other hand during test generation, the path needs to be extended until an observable event occurs.

Interestingly, there is no general correlation between the computation time of MSC verification and test generation. For example, MSC no. 57 in figure 6.9 can be verified comparably fast, whereas its test case generation takes about 5 hours.

Distributed Test Case Processing. Normally, verification of all MSCs on a single machine would have taken about a day; generation of all test cases would have taken

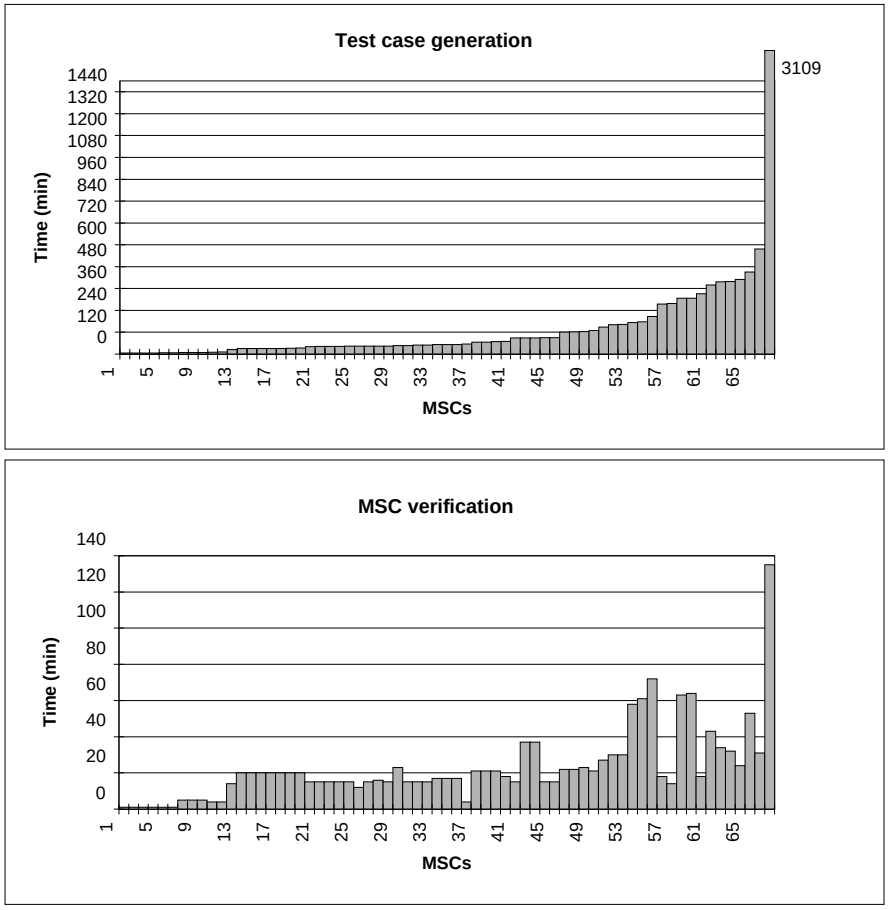


Figure 6.9: Computation time of MSC verifications and test generations

about a week. Therefore, the processing of test purposes was distributed among up to fifteen workstations.

As described in section 6.3, AUTOLINK does not directly write a generated test case into a TTCN MP file. Instead, it stores each test case in an internal representation in memory. This representation and the corresponding constraints can be saved on disk and reloaded later. This feature was used to compute each test case separately. After the computation finished, all test cases were reloaded and combined into a single test suite. Identical constraints were merged automatically during this process.

By means of shell scripts, test generation runs were executed in batch mode such that no manual intervention was needed to start the generation of each single test case. This way, test cases could be generated overnight. Information from previous test generation runs was used to minimize the computation time by placing time-intensive test cases on fast machines first.

6.6.2 VB5.1 and VB5.2

From October 1998 to March 1999, ETSI project STF 116 was concerned with the definition of test purposes and the generation of TTCN-2 test cases for the *real time management coordination* (*RTMC*) protocol used at the VB5.1 reference point. VB5 specifies the physical, procedural, and protocol requirements for interfaces at the reference point between a broadband access network (AN) and a service node (SN). The RTMC protocol of the VB5.1 specification is used to communicate management information about resources in real time between an AN and an SN.

In June 1999, a follow-up project STF 151 was initiated that lasted until March 2000. Its aim was to provide a corresponding test suite for the VB5.2 reference point and its *broadband bearer channel control* (*BBCC*) protocol (ETSI, 1999a). VB5.2 extends VB5.1 by the ability to allocate resources in the AN dynamically under the control of the associated SN.

In STF 151, the same methodology has been applied as in STF 116. According to the ETSI approach, two documents have been created: the TSS&TP document (see ETSI 2000c; 2001a) describes the test suite structure and test purposes; the other document contains the corresponding TTCN-2 test suites for both the AN and SN side (ETSI 2000d; 2001b).

The generation of test purposes and test cases was based on a normative and validated SDL protocol model such that Autolink could be applied. The SDL system consists of two blocks that represent the AN and SN. Unfortunately, the original SDL specification for VB5.2 did not fully specify the PDUs of the BBCC protocol but only those parts that are relevant for the logic of the protocol. For the sake of test generation, it was necessary to extend the specification in such a way that the signals exchanged between AN and SN matched with the official ASN.1 data definitions of the BBCC protocols. In addition, the SDL specification was extended by fault insertion mechanisms to be able to simulate exceptional cases.

A major characteristic of VB5.2 is the large size of its PDUs. A typical TTCN-2 constraint of the VB5.2 test suite is shown in figure 6.10. The readability of the constraints was improved by indenting of nested structures in the *Constraint Value* field but only at the cost of larger constraint declaration tables. This experience has resulted in the development of a prototype for automatic constraint structuring (see section 9.2 on page 153).

MSCs have been produced by interactive simulation. For documentation purposes, large parts of the signal data have been stripped from the MSC test purposes by means of a Perl script. The automatically generated test suites were also post-processed by a Perl script. Among others, test case variables and PIXIT parameters have been introduced afterwards.

ASN.1 PDU Constraint Declaration	
Constraint Name	: cALLOC_14(value : TransIdVal)
PDU Type	: Alloc
Derivation Path	:
Encoding Rule Name	:
Encoding Variation	:
Comments	:
Constraint Value	
{ commonMsgInfo { protDiscr '49'H, transId { transIdLength '03'H, transIdFlag '0'B, transIdVal value }, msgType '40'H, msgCompatInd '80'H, msgLength TSO_MsgLength() }, connRefNoIE { commonIEInfo { iEType '00'H, iECompatInd '80'H, iELength TSO_IELength() }, connRefNoVal PIX_ConnRefNoVal_PtM1 }, aTMTrfcDscrptrIE send : { commonIEInfo { iEType '0A'H, iECompatInd '80'H, iELength TSO_IELength() }, contents PIX_ATM_traffic_descriptor }, brdbndBcapIE { commonIEInfo { iEType '0B'H, iECompatInd '80'H, iELength TSO_IELength() }, contents PIX_Broadband_capability }, qoSParamsIE { commonIEInfo { iEType '0D'H, iECompatInd '80'H, iELength TSO_IELength() }, contents PIX_QOS_parameter }, usrPortConnIdIE { commonIEInfo { iEType '02'H, iECompatInd '80'H, iELength TSO_IELength() }, usrPortConnIdIEOctet5 'A0'H, lgclUsrPortId PIX_LUP_Id1, vpci PIX_LUP_VCPI3, vci PIX_LUP_VCI3 }, srvcPortConnIdIE { commonIEInfo { iEType '03'H, iECompatInd '80'H, iELength TSO_IELength() }, srvcPortConnIdIEOctet5 'A0'H, vpci PIX_LSP_VCPI3, vci PIX_LSP_VCI3 }, branchIdIE { commonIEInfo { iEType '08'H, iECompatInd '80'H, iELength TSO_IELength() }, branchIdVal PIX_PtM1_BranchId1 } }	
Detailed Comments :	

Figure 6.10: A VB5.2 constraint

6.7 Comparison with Other SDL-based Test Tools

In the following, the two test generation tools SAMSTAG and TESTCOMPOSER are sketched. Similar to AUTOLINK, they use SDL for system specification and MSC for test purpose specification.

6.7.1 SaMsTaG

The SAMSTAG (Sdl And Msc baSed Test cAse Generation) method and tool (see, e.g., Nahm, 1995; Toggweiler, 1995) was developed at the University of Berne from 1991 to 1997 and partially funded by SWISSCOM.

In SAMSTAG, an SDL specification is a closed system that describes the complete architecture, including the IUT, the test context, and the tester processes. Test purposes are described in terms of MSCs. In contrast to the approach chosen for AUTOLINK, an MSC test purposes does not have to describe a complete trace with all observable events that must be performed in order to get a pass verdict. Instead, an MSC may only describe an internal signal exchange, an internal action, or the fact that a certain process reaches some specific state.

Based on an MSC test purpose, a TTCN-2 test case is computed in three steps:

1. Computation of *possible pass observables* (*PPOs*)

   By state space exploration, SAMSTAG searches for SDL traces which include the events in the MSCs and drive the SDL system from its initial state back to its initial state. Whenever such a trace is found, all externally visible events are extracted and considered as a PPO.

2. Computation of a *unique pass observable* (*UPO*)

   Since there is no unique relation between an SDL system trace and its observables, i.e., two traces may have the same observable, there is no guarantee that a PPO indeed fulfills the test purpose. Thus, for a given PPO, all SDL traces are simulated which include the events in the PPO. If each trace also fulfills the test purpose, the PPO is considered as a UPO. Only one UPO is needed for test case generation which is chosen from the shortest ones.

3. Computation of *inconclusive observables*

   By yet another state space exploration, all events are determined that lead to an inconclusive verdict. For that purpose, the SDL system is simulated with the UPO determined in the previous step.

A detailed case study for SAMSTAG is presented by Scheurer (1997).

6.7.2 TestComposer

In 1998, VERILOG developed the TESTCOMPOSER tool (Kerbrat et al., 1999) and integrated it into their commercial tool suite OBJECTGEODE. TESTCOMPOSER is based on former experience with the TGV and TVEDA tools which were developed at IRISA/ VERIMAG and France Telecom/CNET (Groz and Risser, 1997).

TESTCOMPOSER follows the overall approach of AUTOLINK and shares many concepts with it. For instance, test purposes are represented by MSCs[6] and TTCN-2 is supported as a common output language. Nevertheless, the two tools differ in many details and put their focus onto different steps of the test generation process. The strength of TESTCOMPOSER is the flexible specification of test purposes while AUTOLINK has its strong points when it comes to the customization of the generated test suites.

TESTCOMPOSER allows to declare an arbitrary block within an SDL system as the SUT. All channels connected to this block become PCOs. For each PCO, the test specifier can define which signals are controllable and observable. Signals that cannot be observed are suppressed in the test case output, whereas uncontrollable signals become *implicit send* events in TTCN. Test purposes may be defined completely (as in AUTOLINK) or partially (similar to SAMSTAG). In the latter case, missing events are added during test generation. In addition, postambles can be computed for paths leading to a pass or inconclusive verdict based on a user-defined goal that characterizes the initial state.

A detailed comparison of AUTOLINK and TESTCOMPOSER is given by Schmitt et al. (2000).

[6] TESTCOMPOSER also creates scripts in a proprietary format that can be handled more efficiently by OBJECTGEODE.

6.8 Discussion

The projects at the European Telecommunications Standards Institute, in particular the Core INAP CS-2 case study, have proven that AUTOLINK is applicable to very complex specifications. Furthermore, the tool allows to generate TTCN test suites with increased quality and a reduced amount of cost and time. It is estimated that expenses were reduced by about 20 percent in the INAP CS-2 project in comparison to manual test specification.

Nevertheless, the use of AUTOLINK has unveiled a few issues that must be considered when using SDL and MSC for automatic test generation. Furthermore, there are different possibilities to enhance the tool that are discussed in the following.

Dynamic Process Creation. A problem that has been identified during the application of AUTOLINK is dynamic process creation at the boundaries of the system. AUTOLINK defines the test architecture based on the static SDL system description. That means, the PCOs are determined by the fixed number of channels to the SDL environment. However, some SDL systems are designed in such a way that several process instances are created dynamically for one process type. These instances share the same statically-defined SDL channels for communication although, in reality, each process instances might have its own set of channels. In this case, the SDL system must be modified such that each process instance and its communication links are specified explicitly. Of course, this is only possible if the number of process instances is fixed and known in advance.

Defaults and Inconclusive Events. In order to be able to execute test cases consecutively, each test case must drive the implementation into the initial state. While a suitable postamble can be specified in the MSC test purpose for successful test case execution, there is no way to define a "reset" test sequence for the failure case. The same holds for *inconclusive* branches. Ideally, a separate postamble should be computed for each inconclusive branch. In analogy to the TESTCOMPOSER, the test specifier should be able to describe the idle state by a boolean expression over state variables.

Complex Signal Parameters. While MSC has proven a good choice for the description of system traces, practical experience has shown that signal parameters — in particular when described by ASN.1 — tend to be voluminous. To enhance the readability of MSCs, large data descriptions should better be defined separately, e.g., in the header of an MSC-2000 document. That means, a (non-standardized) reference mechanism should be introduced into MSC similar to one proposed by Grabowski et al. (1995).

Mapping Signals and Data. One of the major drawbacks of SDL with regard to testing is the requirement that all communication is realized by signal exchange. In TTCN-2, there is no corresponding concept. Instead, the tester and the SUT exchange values

of arbitrary data type. Hence, if a PDU is defined in an external ASN.1 module, it needs to be wrapped up in a signal. During test generation, the signal is mapped to an ASN.1 SEQUENCE which includes the PDU as its only parameter. Since the additional embedding causes some overhead and influences the PDU encoding, an option had to be introduced that allows to strip redundant signal definitions for individual signals. During the standardization process for SDL-2000, the removal of the signal concept has been considered but dropped again due to incalculable implications on the language.

Efficient State Computation. Both case studies have shown that the time needed to generate TTCN test cases from MSC test purposes is negligible in comparison to the time needed to set up a suitable SDL specification and to define test purposes. Nevertheless, test case computations of 51 hours are not desirable. In the INAP CS-2 study case, it has turned out that the major bottleneck is the computation of the hash keys needed for the Supertrace algorithm. For that purpose, the global system state is sequentialized and written in a large byte array on which simple operators such as `xor` and `+` are applied to compute the two hash keys. In case of the INAP CS-2 specification, a single SDL state exceeds 100,000 bytes. As a consequence, only 22 states per minute could be explored on average. A significant performance improvement can be achieved by a compositional computation of the hash keys, i.e., separate hash values are computed for individual processes, blocks, and channels first and combined afterwards.

Revision of Implementation Relation. SDL abstracts from implementation-specific aspects such as process scheduling. That means, the behavior of an SDL system is defined under consideration of *all* possible schedulers. However, scheduling may influence the behavior of a system. This is not taken into account by current SDL test generation methods. In worst case, an automatic test generation tool like AUTOLINK produces test cases that never lead to a *pass* verdict when executed on a peculiar yet correct implementation. In order to ensure that only valid test cases are generated, it must be verified that implementation-specific aspects do not have any effect on test execution. In order to perform these checks, a new state space exploration algorithm needs to be developed. Unfortunately, such an algorithm is expected to make the state space explosion problem worse.

7 Test Generation for Distributed Test Architectures

Most SDL- and MSC-based test generation methods and tools produce test cases for monolithic testers in which a single process controls and observes the entire system under test. The use of such test cases becomes problematic if the SUT is a distributed system with components at different locations. In that case, the test equipment itself has to be a distributed system.

The implementation of a non-concurrent abstract test case for distributed test equipment is a complicated and error-prone task and requires a substantial amount of work. On the other hand, the test languages presented in chapter 3 already provide concepts for describing test components that are executed in parallel. Thus, a method has been developed and implemented in AUTOLINK that allows to generate concurrent TTCN-2 test cases directly from SDL system specifications and MSC test purposes.[1]

This chapter is structured in the following manner: In section 7.1, the major concepts of concurrency in TTCN-2 are recapitulated and some theoretical aspects of synchronization among test components in TTCN-2 are discussed. (A complete description of TTCN-2 can be found in section 3.1.) The generation of distributed test cases requires additional information that cannot be retrieved from an SDL specification or from MSC test purposes. In section 7.2, the necessary information for setting up a test component configuration is listed and a generic class of configurations is presented. Different ways of describing and handling synchronization among different test components in the context of automatic test generation are discussed in section 7.3. In section 7.4, algorithms for the generation of concurrent test cases are described. Finally, their application is demonstrated by two examples in section 7.5.

7.1 Concurrency in TTCN-2

Test Component Configuration. In TTCN-2, a distributed test system is structured into a *main test component* (*MTC*) and one or more *parallel test components* (PTCs). The MTC is responsible for the creation of the PTCs and the computation of the final test verdict.

[1] The methods and algorithms presented in this section have been published before by the author and two colleagues as (Grabowski et al., 1999). The original paper has been revised and restructured for better integration in this thesis.

Communication with the SUT takes place at *points of control and observation* (*PCOs*). Each PCO is assigned exclusively to one test component. Coordination among two test components can be performed by asynchronous exchange of *coordination messages* (*CMs*) at *coordination points* (*CPs*). Communication at PCOs and CPs is bidirectional and asynchronous, i.e., a PCO/CP is modeled by two infinite FIFO buffers.

Even if no CP is specified explicitly between the MTC and a PTC, two types of implicit communication take place:

- The PTC contributes its test verdict to the global result variable which is used by the MTC to compute the final test verdict.
- The PTC informs the MTC about its termination.

The relations between the MTC, the PTCs and the SUT are described by a *test component configuration*. A test suite may contain several test component configurations. For each test case, one of them has to be chosen.

Behavior Description. While the behavior of an MTC is described in a *Test Case Dynamic Behaviour* table, PTCs are specified as test steps. Their descriptions may be given as *local trees* within the behavior table of the MTC or in separate *Test Step Dynamic Behaviour* tables.

An example of a concurrent TTCN-2 test case is shown in figure 7.11 on page 118. The behavior of the MTC is specified by the main behavior tree. The PTC behavior descriptions are included as local trees. The MTC creates all PTCs by calling the corresponding behavior descriptions with the `CREATE` construct (line 1). The termination of the PTCs is checked by means of `DONE` events (lines 7 and 11).

Communication among test components is treated in the same way as communication with the SUT. In figure 7.11, statement *MCP_2!Proceed(Sync2,Y)* on line 6 denotes the sending of CM *Proceed(Sync2,Y)* via CP *MCP_2*. The corresponding receive event is given in line 24.

Synchronization of Test Components. The synchronization of test components may take place implicitly or explicitly. *Implicit synchronization* is performed at the beginning and the end of a test case execution: The MTC creates all PTCs and checks their termination. *Explicit synchronization* can be performed by exchanging CMs between test components.

Coordination messages can be used to coordinate the actions of test components controlling different PCOs. But they cannot ensure the correct order of send and receive test events at different PCOs in all cases. This is due to the asynchronous communication mechanism of TTCN-2 and can be explained by means of a simple example:

Given two test components, *TC1* and *TC2*, controlling different PCOs. *TC1* sends ASP *M1* to the SUT and, in response, the SUT sends ASP *M2* to *TC2* (see figure 7.1). There are two conceivable strategies for ensuring the correct order of the sending of *M1* by *TC1* and the reception of *M2* by *TC2*:

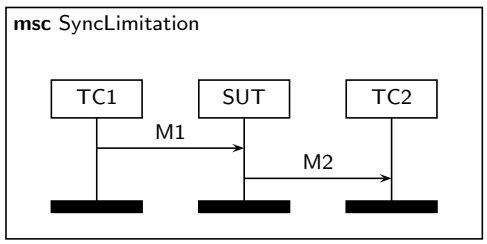


Figure 7.1: Limitations of synchronization

1. *TC1* sends a coordination message *CM1* to *TC2* to indicate that it has sent *M1* and *M2* is the next message to be received from the SUT.
2. *TC2* knows that *M2* is the reaction to *M1* sent by *TC1* and therefore sends a coordination message *CM2* to *TC1* as a request to send *M1*.

In the first case, *M2* may overtake *CM1* and *TC2* will interpret this as a failure, although the actual order was correct. In the second case, neither *TC1* nor *TC2* can decide whether the SUT sends *M2* during the transmission of *CM2* (i.e., before *M1* has been sent), and an incorrect order will pass the test. Thus, neither of the strategies can be used to ensure the correct order of sending *M1* and receiving *M2*. Additional knowledge about the transmission time of messages does not help either. According to the CTMF, "the relative speed of systems executing the test case should not have an impact on the test result".

7.2 Definition of Test Component Configurations

An SDL specification defines the functionality of a system by describing its dynamic behavior. Structural concepts can be used to specify a hierarchical system architecture with regard to logical or implementation aspects. In general, however, the concrete architecture of a system implementation, including the distribution of the different components, cannot be derived from an SDL specification. Whether two blocks or processes are executed on the same machine or on several computers at different locations is outside the scope of the SDL semantics.

By convention, each block in an SDL system diagram could be considered as an entity that has to be controlled and observed by a separate test component. But this convention might impose strong restrictions on the use of SDL's structural concepts. Since it is impossible to determine an appropriate test configuration by the analysis of an SDL specification, additional information has to be provided concerning:

- the test components and their roles (either MTC or PTC),
- the assignment of PCOs to test components (i.e., the connections between the test components and the SUT),
- the coordination points (CPs), and

- the assignment of CPs to test components (i.e., the connections among test components).

This information may be expressed in a graphical form, e.g., in form of a separate SDL system diagram or by a UML deployment diagram. It is also conceivable to provide it already in form of TTCN-2 tables or in a tool-specific command language.

A test generation tool may also be able to define a default configuration automatically, e.g., a test configuration where each PTC handles one PCO only and the MTC is responsible for PTC creation, synchronization, and computation of the final test verdict. (In fact, this is the approach that has been realized in AUTOLINK.)

For the automatic generation of TTCN-2 test suites from SDL system specifications and MSC test purposes, it is reasonable to restrict the set of all possible test configurations. In the following, a generic class of test configurations is considered that is characterized by four properties:

- Each PTC handles at least one PCO.
- Each PTC is connected directly to the MTC by a CP.
- The MTC controls and synchronizes the PTCs but it may also handle PCOs on its own optionally.
- The PTCs do not communicate directly among each other but only indirectly via the MTC.

7.3 Synchronization of Test Components

As mentioned above, the synchronization of test components is an important issue that must be taken into account when specifying test purposes. In accordance with TTCN-2, two types of synchronization are considered: implicit and explicit synchronization.

7.3.1 Implicit Synchronization

Implicit synchronization is carried out at the start and may be carried out at the end of a test case by the MTC. Obviously, the corresponding `CREATE` constructs and `DONE` events can be added to a test case automatically by a test generation tool and do not have to be specified in a test purpose.

Further synchronization is needed if the test system has to guarantee that the first send event happens only after the creation of *all* PTCs. In this case, one of the explicit synchronization mechanisms has to be used.

7.3.2 Explicit Synchronization

During the execution of a concurrent TTCN-2 test case, a test component is not aware of the state of other test components. A lack of synchronization may lead to problems

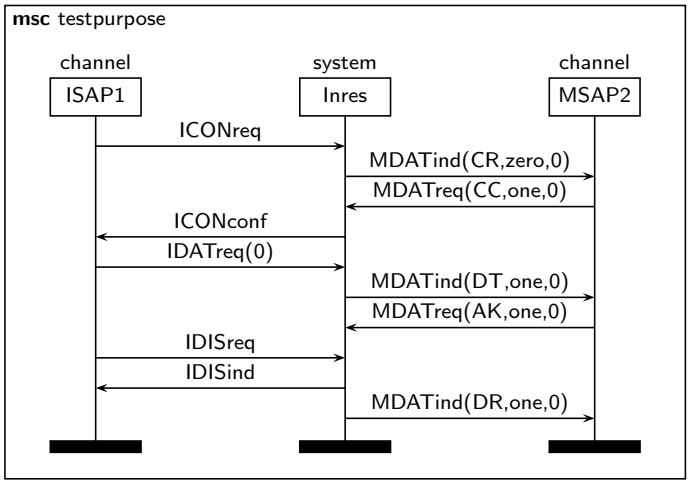


Figure 7.2: Inres test purpose with lack of synchronization

during test execution as illustrated by the MSC given in figure 7.2. According to the MSC semantics, the sending of *IDISreq* by *ISAP1* may happen before, between, or after the reception of *MDATreq(DT,one,0)* and the sending of *MDATreq(AK,one,0)* by *MSAP2*. If *ISAP1!IDISreq* shall always be executed only after the data transfer has been completed, and if *ISAP1* and *MSAP2* are handled by different test components, the latter have to exchange CMs.

Obviously, the points in the control flow of the test components at which such a synchronization must take place cannot be calculated automatically, because they depend on the intention of the test specifier. Without additional information, a test generation tool does not know which ASP shall be sent first or whether the order of events is relevant at all. As a consequence, the synchronization of test components has to be defined explicitly by the test specifier in MSC test purposes.

Depending on the number of test components involved and test events to be coordinated, the CM exchange which is necessary for an explicit synchronization may become very complex. To cope with simple as well as complex situations, two means for the specification of explicit synchronization are considered:

- Explicit description of the CM exchange by MSC messages
- Definition of synchronization points by MSC conditions

In the latter case, the concrete exchange of CMs among the test components is derived automatically.

7.3.2.1 Synchronization by Coordination Messages

The definition of explicit synchronization by describing the exchange of CMs in MSC test purposes is not as trivial as it seems to be at first glance. The reason is that in MSCs used for test purpose description, the instances represent PCOs and not the test

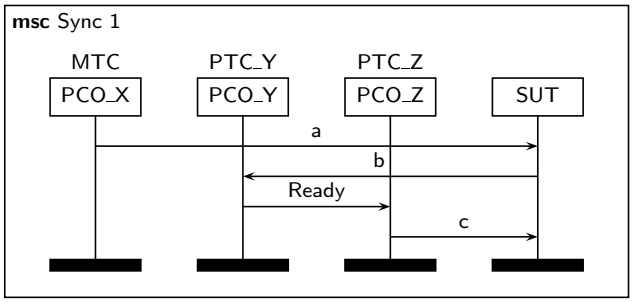


Figure 7.3: Explicit synchronization by means of a coordination message

components controlling these PCOs.[2] On the other hand, CMs are exchanged between test components. Neither the actual sender and receiver of a CM nor the coordination points involved are represented in the MSC test purposes.

Nevertheless, CMs between PCO instances can be specified and interpreted as follows: A coordination message *CM1* with its origin at PCO instance *PCO_A* and its target at PCO instance *PCO_B* coordinates test events at *PCO_A* and *PCO_B*. The origin refers to the send event of *CM1* by the test component controlling *PCO_A*. The target refers to the corresponding receive event by the test component controlling *PCO_B*. The order of events (including send and receive events of CMs) along a PCO instance has to be preserved by the test component controlling the PCO.

It should be noted that no (graphical) distinction between CMs and ASPs has to be made in MSC test purposes. Origin and target of a CM arrow are PCO instances. For ASPs, either the origin or the target has to be the SUT instance.

Figure 7.3 shows an MSC test purpose description. The mapping of PCOs to test components is specified in the MSC instance headers, e.g., *PCO_X* belongs to the MTC. The MSC includes the CM *Ready* added manually by the test specifier. It is interpreted as follows: *PTC_Y* shall send the CM *Ready* after the reception of ASP *b* at *PCO_Y*, and *PTC_Z* shall send ASP *c* after the reception of CM *Ready*. That means that CM *Ready* forces test component *PTC_Z* to postpone the execution of $PCO\_Z!c$ until *PTC_Y* has received *b*.

The manual specification of CMs becomes difficult if more than two test components are involved, and if CM receive events are alternatives to receptions of ASPs.

Figure 7.4 shows another test purpose example where the correct specification of a coordination is less simple. It is interpreted as follows: First, *PTC_X* sends ASP *a* via *PCO_X* to the SUT. The system under test in turn answers with ASPs *b*, *c*, and *d* that are to be observed at *PCO_Y* and *PCO_Z*, respectively. ASP *e* should be sent via *PCO_Z* only after the reception of *b*, *c*, and *d*. To ensure this, the reception of *b* has to be confirmed by means of CM *Ready* sent by *PTC_Y*.

ASPs *c* and *d* are received via the same FIFO queue *PCO_Z*, and their order is given by the test purpose. CM *Ready* is received by *PTC_Z* via a second FIFO queue, i.e., the CP

[2]Note that a test component may control and observe more than one PCO.

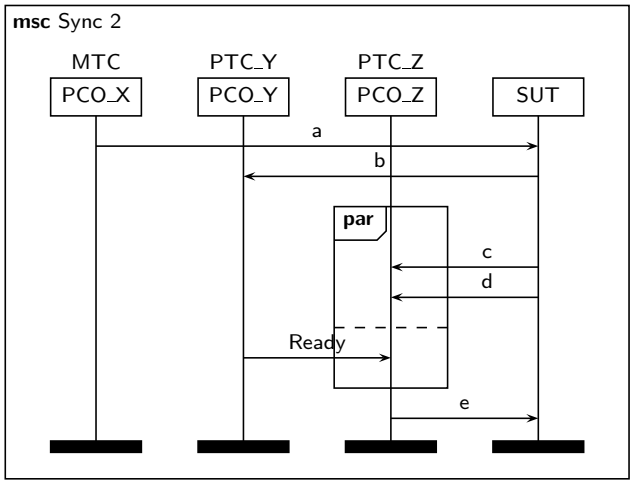


Figure 7.4: Coordination messages – Complex example

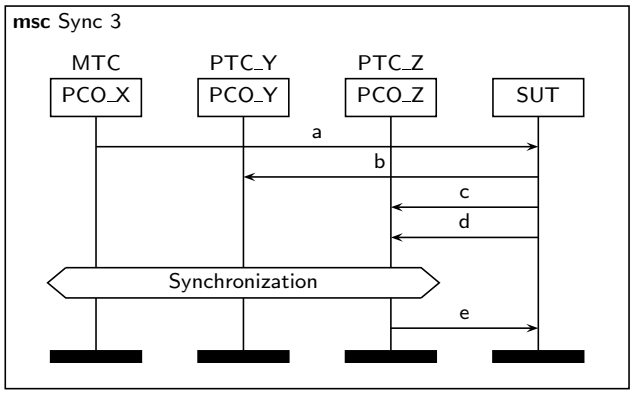


Figure 7.5: Explicit synchronization by means of an MSC condition

between *PTC_Y* and *PTC_Z*. It cannot be predicted if the CM *Ready* is received before, after or between the reception of *c* and *d*. Thus, a parallel operator (or, alternatively, a co-region with general ordering) has to be used to specify all possible orders of reception.

Obviously, the manual drawing of CMs becomes more and more complicated if the number of test components, CMs, and CPs involved increases. In order to ease test specification, another possibility to describe explicit synchronization is presented in the next section.

7.3.2.2 Synchronization by MSC Conditions

MSC conditions are a simple yet robust and consistent means to specify synchronization. Conditions used for test synchronization purposes only cover PCO instances. These *synchronization conditions* define common *synchronization points* within the message flow at the different PCOs.

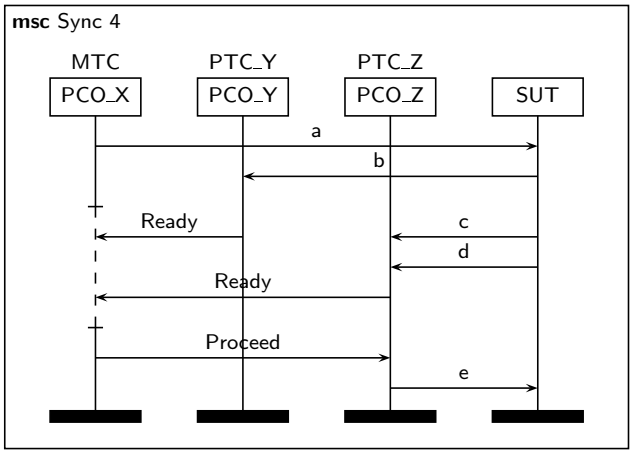


Figure 7.6: Automatically generated CM exchange for the condition in figure 7.5

Figure 7.5 shows an MSC test purpose description with a synchronization condition. The desired effect of the synchronization is the same as in figure 7.4: *PTC_Z* shall not send ASP *e* before the ASPs *b*, *c* and *d* have been received.

During test case generation, the synchronization conditions are used to compute the actual exchange of CMs between test components. There are several possibilities to perform a synchronization by means of message exchange. For implementation in AUTOLINK, it has been decided to support a mechanism which allows to synchronize the ASP exchange of an arbitrary number of PCOs. The synchronization is managed by the MTC, and the principle of the mechanism is simple:

As already stated above, synchronization conditions define common synchronization points within the ASP exchange at different PCOs. A PCO can be seen as a sequential process, and the test component handling the PCO as the process manager. If a PCO reaches a synchronization point, this is reported to the MTC and the PCO enters a *waiting* state. If the test event following the synchronization point is the reception of an ASP, the PCO simply waits for this ASP. If it is a send event, the PCO has to wait for a CM from the MTC to get the permission to send the next ASP to the SUT. This mechanism ensures that all PCOs involved in the synchronization reach the synchronization point before any new ASP is sent at one of these PCOs.

By applying this approach, the CM exchange shown in figure 7.6 is generated automatically for the condition in figure 7.5. The *Ready* CMs are used to indicate that the synchronization point has been reached, and CM *Proceed* is used to trigger the sending of ASP *e*.

7.4 A Test Generation Procedure

In this section, algorithms are presented that allow to derive a concurrent TTCN-2 test case from an SDL specification and an MSC test purpose where MSC conditions are used as a means of synchronization.

7.4.1 Simulator Requirements

The test generation algorithm is based on the basic functionality provided by a general purpose state space exploration tool which allows the combined simulation of an SDL specification and an MSC test purpose (e.g., Grabowski et al., 1993; Telelogic (formerly Verilog), 2002; Telelogic, 2002a). In particular, the following two functions must be available:

- $state_{sim}$.nextEvents()

  Given a state $state_{sim}$, nextEvents returns the set of all events which may occur next. $state_{sim}$ describes both the current global state of the simulated SDL system (i.e., the state, timers, and variable values of each individual process, queue contents etc.) and the progress in the MSC (i.e., the events which take place next at each instance). If nextEvents returns the empty set, the MSC is supposed to be verified completely, i.e., a path through the reachability graph of the SDL system has been found which satisfies the MSC.[3]

- $state_{sim}$.nextState(e)

  Given a state $state_{sim}$ and an event e, nextState returns the state which is obtained if e is executed in $state_{sim}$.

There are several different types of events that might happen during simulation. Some of them may only refer to the SDL system (e.g., internal events that are not represented in the MSC), while others may only refer to the MSC (e.g., events related to synchronization conditions). With regard to the generation of test cases for distributed test architectures, four types of events are considered:

- Event '*Send from SDL environment*' ($pco\,!\,sig$)

  ASPs sent from the environment into the SDL system during simulation become TTCN send events. To be able to specify a send event, the simulator is required to return both the complete signal and the channel (PCO) through which the ASP was sent.

- Event '*Send to SDL environment*' ($pco\,?\,sig$)

  ASPs sent by the SDL system to its environment become receive events in the TTCN test case. In analogy to send events, the simulator has to report both the ASP and the PCO.

- Event '*Enter synchronization*' (enterSync(*id*, *pco*, *Cur*, *All*))

  Whenever an instance in the MSC test purpose reaches a synchronization condition, a special event, called *enterSync*, has to be returned by function nextEvents. *enterSync* has four parameters: *id* denotes the unique identifier of the condition; *pco* is the name of the PCO instance which has reached the synchronization condition; *Cur* is the set of instances that have reached the condition so far; and *All* denotes the set of all instances which are involved in the synchronization.

[3] For simplicity, the cases that a deadlock occurs or the simulation is stopped, because the behavior of the SDL system does not comply with the MSC, are neglected.

Algorithm 7.1 Invocation of the test generation for the PTCs and the MTC

```
1:  testgen() {
2:    for all i ∈ {1, ..., n} {
3:      root_ptc_i := newStartNode();
4:      testgenPTC( i, root_sim, root_ptc_i );
5:    }
6:    root_mtc := newStartNode();
7:    testgenMTC( root_sim, root_mtc.addTransition( CREATE( PTC_1:TestPTC_1, ... ,
8:                                                          PTC_n:TestPTC_n ) ) );
9:  }
```

- Event '*Leave synchronization*' (leaveSync(*id*, *pco*))

  When all instances engaged in a synchronization have entered the condition, the simulator skips the condition. For each instance in the MSC which sends a message directly after the condition, a notification is issued by nextEvents to indicate that the message is allowed to be sent now. To find out whether such an event has to be created by the simulator engine, an initial static analysis of the MSC is sufficient.

All other events which might be reported by the simulator engine are skipped during test generation. This means that they are executed in order to get to the next system state but they are not transformed into TTCN events.

7.4.2 Test Generation for MTC and PTCs

There are two approaches to generate a distributed test case based on a given SDL specification and an MSC test purpose: On the one hand, a behavior tree may be generated first which covers all signals exchanged between the tester and the SUT, plus additional information about synchronizations. Based on such a complete test description, behavior trees for the MTC and all PTCs can be extracted. Alternatively, separate behavior trees for the MTC and the PTCs may be created immediately at the time of simulation of the SDL specification.

The test generation algorithms for both approaches are basically the same. For better comprehension, a solution is presented for the latter approach in this thesis. For AUTOLINK, the first alternative has been chosen, because it allows for generating test suites for different test component configurations without having to repeat the state space exploration.

The algorithms 7.1, 7.2, and 7.3 describe the construction of the dynamic behavior trees. Algorithm 7.1 is the main function that invokes the actual test generation functions. The algorithms 7.2 and 7.3 for the MTC and the PTCs are structurally similar. To a certain extend, the algorithm for the MTC is the inverse of the algorithm for the PTCs. For example, if a PTC sends a coordination message to the MTC, a corresponding receive

Algorithm 7.2 Test generation algorithm for the MTC

```
1:  testgenMTC(state_sim, state_test) {
2:    E := state_sim.nextEvents();
3:    if ( E = ∅ ) {
4:      nextstate_test := state_test.addTransition( ?DONE( PTC_1, ..., PTC_n ) );
5:    } else if ( ∃e ∈ E : e = pco!sig ∧ pco ∈ PCO_MTC ) {
6:      nextstate_test := state_test.addTransition( e );
7:      testgenMTC( state_sim.nextState( e ), nextstate_test );
8:    } else if (∃e ∈ E : e = leaveSync( id, pco ) ∧
9:                ∃i ∈ {1,...,n} : pco ∈ PCO_i) {
10:     nextstate_test := state_test.addTransition( MCP_i!Proceed( id, pco ) );
11:     testgenMTC( state_sim.nextState( e ), nextstate_test );
12:   } else {
13:     for all e ∈ E {
14:       if ( e = pco?sig ∧ pco ∈ PCO_MTC) {
15:         nextstate_test := state_test.addTransition( e );
16:       } else if ( e = enterSync( id, pco, Cur, All ) ∧
17:                   ∃i ∈ {1,...,n} : pco ∈ PCO_i ∧ (PCO_i ∩ All) ⊆ Cur ∧
18:                   All ⊈ PCO_i) {
19:         nextstate_test := state_test.addTransition( MCP_i?Ready( id ) );
20:       } else {
21:         nextstate_test := state_test;
22:       }
23:       testgenMTC( state_sim.nextState( e ), nextstate_test );
24:     }
25:   }
26: }
```

event has to be added to the MTC behavior description. In addition, `CREATE` statements and `DONE` events have to be added at the root and the leaves of the MTC behavior tree (lines 7 and 8 in algorithm 7.1 and line 4 in algorithm 7.2). Due to the similarity of both algorithms, only the algorithm for PTCs is described in the following.

For the construction of the behavior tree, two functions are applied:

- newStartNode()

  This function returns an (empty) behavior tree with a single state node and no test events.

- $state_{test}$.addTransition(e)

  If an outgoing edge from $state_{test}$ labeled with e exists for a given state $state_{test}$ in the behavior tree and a test event e, addTransition simply returns the successor node; otherwise a new node $state_{next}$ is added to the behavior tree with an edge from $state_{test}$ to $state_{next}$ labeled with e, and $state_{next}$ is returned.

Algorithm 7.3 Test generation algorithm for PTC_i

```
1:  testgenPTC( i, state_sim, state_test ) {
2:    E := state_sim.nextEvents();
3:    if ( ∃e ∈ E : e = pco ! sig ∧ pco ∈ PCO_i ) {
4:      nextstate_test := state_test.addTransition( e );
5:      testgenPTC( i, state_sim.nextState( e ), nextstate_test );
6:    } else if ( ∃e ∈ E : e = enterSync( id, pco, Cur, All ) ∧
7:                  pco ∈ PCO_i ∧ (PCO_i ∩ All) ⊆ Cur ∧ All ⊈ PCO_i ) {
8:      nextstate_test := state_test.addTransition( MCP_i ! Ready( id ) );
9:      testgenPTC( i, state_sim.nextState( e ), nextstate_test );
10:   } else {
11:     for all e ∈ E {
12:       if ( e = pco ? sig ∧ pco ∈ PCO_i ) {
13:         nextstate_test := state_test.addTransition( e );
14:       } else if ( e = leaveSync( id, pco ) ∧ pco ∈ PCO_i) {
15:         nextstate_test := state_test.addTransition( MCP_i ? Proceed( id, pco ) );
16:       } else {
17:         nextstate_test := state_test;
18:       }
19:       testgenPTC( i, state_sim.nextState( e ), nextstate_test );
20:     }
21:   }
22: }
```

In the following, the PTCs are supposed to be numbered ($i \in \{1, \ldots, n\}$). The set of all PCOs which belong to parallel test component PTC_i is defined as PCO_i.

The PTC algorithm (algorithm 7.3) is invoked with three parameters: i denotes the number of the PTC; $state_{sim}$ is the current node in the reachability graph of the SDL system (initially the root node; see line 4 in algorithm 7.1) and $state_{test}$ is the current node in the behavior tree to be constructed.

At first, all possible next events are requested from the simulator engine by function nextEvents (line 2 in algorithm 7.3). Then it is checked whether the PTC can send either an ASP to the SUT (line 3) or a coordination message to the MTC (lines 6 and 7). Whenever a test component can send a signal, it should do so immediately. In that case, no alternatives are taken into account. Instead, the send event is added to the behavior tree (lines 4 and 8) and the algorithm is invoked recursively with the successor node of $state_{sim}$ and $nextstate_{test}$ (lines 5 and 9). Only if the PTC cannot send a signal, all (remaining) events have to be considered, as indicated by the loop in lines 11–20.

If an event has to be appended to the behavior tree, a transition to a new node ($nextstate_{test}$) is inserted into the tree (lines 4, 8, 13 and 15). Otherwise, $nextstate_{test}$ is set to $state_{test}$ (line 17). By invoking itself recursively, testgenPTC explores the whole

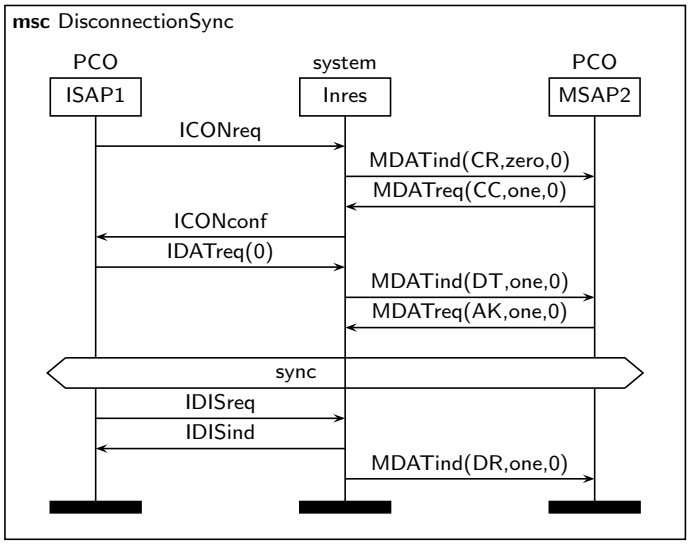


Figure 7.7: MSC *DisconnectionSync*

state space of the SDL/MSC specification. Due to the interleaving semantics of SDL, the algorithm might reach a state where it wants to add an already existing edge to the behavior tree. As described above, this case is captured by function addTransition.

There are two CMs used for the communication between a PTC and the MTC: CM *Ready*(*id*) is sent from a PTC to the MTC in order to indicate that all relevant instances of the PTC have reached synchronization condition *id* (line 8). It is only sent if:

1. an *enterSync* event is reported by the simulator engine,
2. all instances of the PTC which are involved in the synchronization have already reached the condition $((PCO_i \cap All) \subseteq Cur)$, and
3. there are other test components which are also involved in the synchronization $(All \not\subseteq PCO_i)$.

In reverse, CM *Proceed*(*id*, *pco*) is received from the MTC and indicates that the PTC is allowed to send further ASPs via *pco* (line 15).

7.5 Case Studies

The application of the test generation algorithms is demonstrated by two examples.

A Synchronized Inres Test Purpose. The use of a synchronization condition for the test purpose in figure 7.2 is shown in figure 7.7. Figures 7.8 and 7.9 present the behavior descriptions that were generated automatically by the AUTOLINK tool. AUTOLINK requires that each PCO is handled by a separate PTC. Thus, a test configuration with two PTCs (*PTC_ISAP1*/*PTC_MSAP2*) and an MTC that coordinates the PTCs is assumed.

Test Case Dynamic Behaviour					
Test Case Name : DisconnectionSync					
Group : Transmission/					
Purpose :					
Configuration : StandardConfiguration					
Default : OtherwiseFail					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		CREATE(PTC_ISAP1 : DisconnectionSync_PTC_ISAP1)		(PASS)	
2		CREATE(PTC_MSAP2 : DisconnectionSync_PTC_MSAP2)			
3		+Synchronization			
4		?DONE(PTC_ISAP1, PTC_MSAP2)		R	
		Synchronization			
5		CP_MSAP2 ? CM	Ready_Indication		
6		CP_ISAP1 ? CM	Ready_Indication		
7		CP_ISAP1 ! CM	Proceed_Indication		
8		CP_ISAP1 ? CM	Ready_Indication		
9		CP_MSAP2 ? CM	Ready_Indication		
10		CP_ISAP1 ! CM	Proceed_Indication		
Detailed Comments :					

Figure 7.8: TTCN-2 test case *DisconnectionSync* – MTC behavior description

A separate test step is defined for the MTC that describes the coordination messages resulting from the synchronization condition. Due to the FIFO semantics of PCOs and CPs, this approach is sound also for the case that events related to a particular synchronization interleave with other events, e.g., events of other synchronizations or message exchanges with the SUT via some PCO. However, if test execution fails, there might be unprocessed events in the FIFO queues that are not logged by the test system and make it more difficult to analyze the cause of the failure.

Multiple Synchronization Conditions. In figure 7.10, an MSC test purpose with two synchronization conditions covering different sets of instances is shown. For this MSC, a test configuration with two PTCs and an MTC is assumed. *PTC_1* controls two different PCOs (*PCO_W* and *PCO_X*); *PTC_2* communicates with the SUT via *PCO_Y*. The MTC exchanges signals both with the PTCs via *MCP_1* and *MCP_2* and with the SUT via *PCO_Z*.

Figure 7.11 shows the behavior descriptions of the MTC and the two PTCs (named *Test_PTC_1* and *Test_PTC_2*). As can be seen, only one synchronization message is sent to the MTC for each PTC. The size of an MTC behavior description mainly depends on the number of PTCs which are involved in a synchronization, since the MTC must be able to receive *Ready* CMs in every possible order. As described above, a simple way to minimize the size of the behavior description is to define a separate test step for each synchronization.

Test Step Dynamic Behaviour					
Test Step Name : DisconnectionSync_PTC_ISAP1					
Group : ParallelTestComponents/Transmission					
Objective :					
Default : OtherwiseFail					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		ISAP1 ! ICONreq	Connection_Request		
2		ISAP1 ? ICONconf	Connection_Confirmation		
3		ISAP1 ! IDATreq	Data_Request(TestSuitePar)		
4		CP_ISAP1 ! CM	Ready_Indication		
5		CP_ISAP1 ? CM	Proceed_Indication		
6		ISAP1 ! IDISreq	Disconnection_Request		
7		ISAP1 ? IDISind	Disconnection_Indication	PASS	
8		ISAP1 ? IDISind	Disconnection_Indication	INCONC	
Detailed Comments :					

Test Step Dynamic Behaviour					
Test Step Name : DisconnectionSync_PTC_MSAP2					
Group : ParallelTestComponents/Transmission					
Objective :					
Default : OtherwiseFail					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		MSAP2 ? MDATind	Medium_Connection_Request		
2		MSAP2 ! MDATreq	Medium_Connection_Confirmation		
3		MSAP2 ? MDATind	Medium_Data_Transfer		
4		MSAP2 ! MDATreq	Medium_Data_Acknowledgment		
5		CP_MSAP2 ! CM	Ready_Indication		
6		MSAP2 ? MDATind	Medium_Disconnection_Request	PASS	
7		MSAP2 ? MDATind	Medium_Data_Transfer	INCONC	
8		MSAP2 ? MDATind	Medium_Connection_Request	INCONC	
Detailed Comments :					

Figure 7.9: TTCN-2 test case *DisconnectionSync* – PTC behavior descriptions

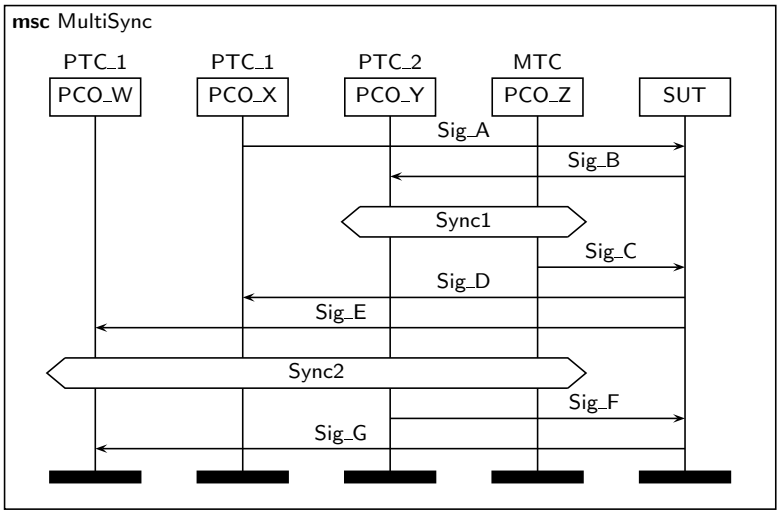


Figure 7.10: MSC *MultiSync*

Test Case Dynamic Behaviour					
Test Case Name : MultiSync					
Group :					
Purpose :					
Configuration : Conf					
Default :					
Comments: : The tester consists of a main test component and two parallel test components					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		CREATE(PTC_1 : Test_PTC_1, PTC_2 : Test_PTC_2)			
2		MCP_2 ? Ready	C_Ready(Sync1)		
3		PCO_Z ! Sig_C	C_Sig_C		
4		MCP_1 ? Ready	C_Ready(Sync2)		
5		MCP_2 ? Ready	C_Ready(Sync2)		
6		MCP_2 ! Proceed	C_Proceed(Sync2, Y)		
7		? DONE(PTC_1, PTC_2)		R	
8		MCP_2 ? Ready	C_Ready(Sync2)		
9		MCP_1 ? Ready	C_Ready(Sync2)		
10		MCP_2 ! Proceed	C_Proceed(Sync2, Y)		
11		? DONE(PTC_1, PTC_2)		R	
		Test_PTC_1			
12		PCO_X ! Sig_A	C_Sig_A		
13		PCO_X ? Sig_D	C_Sig_D		
14		PCO_W ? Sig_E	C_Sig_E		
15		MCP_1 ! Ready	C_Ready(Sync2)		
16		PCO_W ? Sig_G	C_Sig_G	(P)	
17		PCO_W ? Sig_E	C_Sig_E		
18		PCO_X ? Sig_D	C_Sig_D		
19		MCP_1 ! Ready	C_Ready(Sync2)		
20		PCO_W ? Sig_G	C_Sig_G	(P)	
		Test_PTC_2			
21		PCO_Y ? Sig_B	C_Sig_B		
22		MCP_2 ! Ready	C_Ready(Sync1)		
23		MCP_2 ! Ready	C_Ready(Sync2)		
24		MCP2 ? Proceed	C_Proceed(Sync2, Y)		
25		PCO_Y ! Sig_F	C_Sig_F	(P)	
Detailed Comments :					

Figure 7.11: Concurrent TTCN-2 test case *MultiSync*

8 The Tree Walk Search Strategy

The main problem of verification, validation, and test generation is the inherent complexity of most protocols and applications. This complexity manifests itself in a behavior tree that grows exponentially with increasing depth. Even for simple, academic protocols, the state space can be too large to be explored completely. Most specification languages even have language concepts that imply an infinite state space. For instance, incoming signals of an SDL process are handled by a FIFO queue with infinite capacity.

In order to cope with the state space explosion problem, various reduction techniques have been proposed in literature such as partial order simulation, symmetry analysis, symbolic model checking, compositional verification, and data flow analysis.[1] Some approaches aim at ignoring those parts of the state space (states or transitions) that are irrelevant for test generation or proving a specific system property. Others try to simplify the specification (abstraction) or consider its modules first before analyzing the system as a whole. Again other techniques allow to represent the complete state space by an efficient data structure. In addition to theoretically sound methods, heuristics can be used.

In spite of these approaches, the remaining part of the state space cannot be explored completely in practice. Thus in the test purpose specification phase, the *order* in which it is traversed is very important, because it has a major impact on which coverage can be achieved in the available period of time. The selected *search strategy* also determines the size and structure of a test suite. E.g., some strategies tend to create large traces with redundant – from a test specifier's point of view "senseless" – events that do not contribute to fulfilling the test purpose.

In practice, the requirements on a good search strategy for test generation, i.e., getting a high test coverage quickly and finding short, reasonable traces, are partially contradictory and hence a good compromise between both requirements is needed.

In this chapter, a new search strategy, called Tree Walk, is presented. It combines the abilities to find short, reasonable test sequences and to examine deeply hidden parts of the behaviour tree. Tree Walk is a deterministic algorithm that performs a sequence of tree searches with increasing depth, starting at various states in the behavior tree. It is based on the heuristic that in a region of the state space where an increase of the system coverage has been observed, it is likely to detect further transitions that result in an even higher coverage.

In section 8.1, a few traditional search strategies are presented and reasons are given for why they are inadequate for efficient test generation. To formalize some important

[1] An overview of existing techniques with focus on petri nets is given by Rauhamaa (1990).

aspects of Tree Walk, the concept of *labeled transition systems* (*LTS*) is introduced in section 8.2. The main concepts of Tree Walk and an algorithmic description are given in section 8.3. For efficiency, a search strategy should always be combined with some state space reduction techniques. In section 8.4, a new hash algorithm is presented that allows to prune search paths under certain circumstances if a state is revisited. The superiority of Tree Walk to iterative depth-first search and Random Walk is demonstrated by two cases studies in section 8.5. Finally, future enhancements are discussed in section 8.6.

8.1 Classical Search Strategies

Depth-First and Breadth-First Search. Two general-purpose search strategies that are widely used are depth-first and breadth-first search. A *depth-first search* (*DFS*) is characterized by following one path through the behavior tree before considering alternatives. I.e., in every state, one of the possible next events is executed and the search algorithm is invoked recursively with the successor state. Alternative events are considered only during back-tracking. In the context of telecommunication protocols, most systems do not terminate and an unbounded DFS will analyze only one large path. Therefore, the user typically specifies a maximum search depth in which the exploration is stopped.

A *breadth-first search* (*BFS*) solves the problem of infinite paths. For any state s, first all possible alternative states (with regard to s) are investigated before the successor states of s are examined. One of the drawbacks of a BFS is that it must keep track on which states have been visited so far. Therefore, an *iterative DFS* is used as an approximation of a BFS, i.e., a sequence $DFS(k_1), DFS(k_2), \ldots$ of k_i-bounded DFSs is started where $k_i = init + (i-1) * inc$ with *init* being the initial maximum search depth and *inc* being the increment for each iteration.

When used for test generation purposes, breadth-first search and iterative depth-first search result in short test cases but due to the state space explosion problem it is impossible to find test cases for test purposes that require long sequences of transitions. On the other hand, a loosely bounded depth search is more likely to achieve a good coverage but the resulting test cases tend to include initial sequences of non-reasonable events, as the system is driven accidentally into a desired state.

(Guided) Random Walk. The former search strategies are deterministic, i.e., the exact order in which the transitions in the behavior tree are traversed is predictable. In contrast, the *Random Walk* search strategy, as the name indicates, explores the state space indeterministically. Starting at the start state of the behavior tree, a random transition is selected repetitively until either an end state (deadlock) is reached or some limiting heuristic applies, e.g., a maximum search depth is reached. Thereafter, a new and independent exploration is started from the initial state again.

Random Walk has proven to be an adequate search strategy for validation purposes as it allows to uncover efficiently many violations of the dynamic semantics of a specification (West, 1992).

In literature, several proposals have been made to improve the efficiency of Random Walk. A *Guided Random Walk* classifies all possible next transitions in a given state and chooses the one with highest priority. For instance, Feijs et al. (2000) suggest assigning different probabilities to input and output events. The *Hit-or-Jump* algorithm by Cavalli et al. (1999) combines the Random Walk approach with shallow tree searches; if the coverage does not increase during a k-bounded DFS/BFS, a random walk with k transitions is made and another k-bounded DFS/BFS is started at the end state.

With regard to test generation, the usability of Random Walk is restricted for the same reason why an unbounded depth-first search is inappropriate: The resulting test cases tend to be very large and contain many events which a human reviewer considers neither as an essential part of the test preamble nor as part of the test body.

8.2 Labeled Transition Systems

In the following, some concepts of labeled transition systems are introduced. They are used to formalize complicated aspects of Tree Walk, in particular the extension presented in section 8.4.

Definition 4 (Labeled Transition System) *A* labeled transition system *(*LTS*) is a tuple* $\langle S, L, T, s_0 \rangle$*, where*

- S *is a set of system states,*
- L *is a set of labels (also called* actions*),*
- $T \subseteq S \times L \times S$ *is a set of transitions,*
- $s_0 \in S$ *is the initial state of the LTS.*

LTSs can be used as a formal model for reactive systems (ITU-T, 1997b, pp. 33 and 34). Based on such a model, a behavior tree or a directed, acyclic reachability graph can be constructed. In the following, all LTSs are assumed to be deterministic, i.e., $\forall s, s', s'' \in S, l \in L : (s, l, s') \in T \wedge (s, l, s'') \in T \Rightarrow s' = s''$. Although a reactive system is not necessarily deterministic, this requirement is made to reflect the fact that a simulator must behave deterministically to work properly.

Definition 5 (Notational Conventions) *Given an LTS* $\langle S, L, T, s_0 \rangle$*,* $s, s' \in S$*,* $l \in L$*, and* $n \in \mathbb{N}$*, the following notational conventions shall apply:*

$$
\begin{aligned}
s \xrightarrow{l} s' &\equiv (s, l, s') \in T \\
s \rightarrow s' &\equiv \exists l \in L : (s, l, s') \in T \\
s \rightarrow^n s' &\equiv \exists s_0, \ldots, s_n \in S : \\
&\qquad s = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n = s' \\
s \rightarrow^* s' &:= \exists n \in \mathbb{N} : s \rightarrow^n s' \\
s' \textit{ is reachable from } s &:= s \rightarrow^* s'
\end{aligned}
$$

Definition 6 (Path and Distance) *Given an LTS* $\langle S, L, T, s_0 \rangle$. $p = l_1, l_2, \ldots, l_n$ *with* $l_i \in L$ *is a* path *from state* s *to state* s' *if* $\exists s_0, \ldots, s_n \in S : s = s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} s_n = s'$. *The set of all possible paths for a given LTS is denoted by* $Path := Path_{LTS}$.

The length $|p|$ *of a path* p *is equal to the number of its actions/events.*

Given two paths $p = l_1, l_2, \ldots, l_n$ *and* $p' = l'_1, l'_2, \ldots, l'_m$. p *is called a* subpath of p' *if* $n \leq m \wedge \forall i \in \{1 \ldots n\} : l_i = l'_i$.

The distance $|s, s'|$ *between two states* $s, s' \in S$ *is defined as*

$$|s, s'| := \begin{cases} n & s \rightarrow^n s' \wedge \forall m < n : \neg(s \rightarrow^m s') \\ \infty & \neg(s \rightarrow^* s') \end{cases}$$

Since most systems contain loops or even have an infinite number of states, it is impossible to explore a state space completely. Instead, only a small fragment can be examined.

Definition 7 (Sub-LTS) *Given a labeled transition system* $LTS = \langle S, L, T, s_0 \rangle$. LTS_r^n *denotes the part of the* LTS *that is reachable from root state* $r \in S$ *by* $n \in \mathbb{N}$ *transitions. It is defined as* $LTS_r^n = \langle S^n, L^n, T^n, r \rangle$ *with*

$$\begin{aligned} S^n &:= \{r\} \cup \{s \mid s \in S \wedge \exists k \in \mathbb{N}, k \leq n : r \rightarrow^k s \text{ in } LTS\} \\ T^n &:= \{(s, l, s') \mid (s, l, s') \in T \wedge (s = r \vee \exists k \in \mathbb{N}, k < n : r \rightarrow^k s \text{ in } LTS\} \\ L^n &:= \{l \mid (s, l, s') \in T^n\} \end{aligned}$$

With regard to simulation, LTS_r^n can be explored by an n-bounded DFS starting at root state r.

$LTS_r := LTS_r^\infty$ is that part of the LTS that is reachable from root state r.

The set of states that are reachable in n transitions from state r is defined as $States(LTS, r, n) := \{s \mid r \rightarrow^n s\}$. For LTS_r^n, $States(LTS_r^n, r, n)$ denotes the states in maximum depth. For convenience, $Front(LTS_n^r) := States(LTS_r^n, r, n)$ is used as alternative notation.

8.3 Main Concepts

Tree Walk is a search strategy that aims at achieving a good coverage while exploring only a small fragment of the (possibly infinite) state space. Instead of starting a single (iterative) DFS or BFS at the initial system state, Tree Walk starts a large number of shallow, i.e., strongly-bounded, DFSs at different states. These states are called *root states*. First, only the initial system state is a root state but the set of root states changes dynamically, i.e., root states are added and removed at run-time.

Though the exact criteria for a state becoming a root state is outside the scope of Tree Walk, it is assumed that the state selection is based on some coverage criteria such as

symbol (C0) coverage. If, during state space exploration, a state (or an event leading to some state) is found that results in increased coverage, then this state is added to the set of root states.

Tree Walk is based on the fundamental assumption that it is more likely to find another state with further coverage increase near a root state than in other parts of the state space. This assumption is justified by the fact that many telecommunication systems proceed in different stages. For example, the Inres service involves the three phases *connection establishment*, *data transfer*, and *disconnection*. In this case, each phase can be considered independently from the previous ones.

Tree Walk supports this idea by declaring the state in which the connection is established as root state r. If both stages take at most c and d steps, i.e., the longest path for any connection establishment attempt (whether successful or not) comprises c transitions, then it might be sufficient to explore LTS^{c}_{init} and LTS^{d}_{r} to achieve the same coverage as when exploring LTS^{c+d}_{init}.

8.3.1 Root States

Tree Walk maintains a list of root states that are used as initial states for exploration. A root state is characterized by a tuple $\langle path, depth, status \rangle$ where

- $path \in Path$ is a path from the initial state of system to the root state.
- $depth \in \mathbb{N}$ is the maximum search depth for the next bounded DFS (initially: 1).
- $status \in \{ok, ignore, done\}$ specifies whether the root state is relevant for state space exploration and test generation (see below).

The set of all root states is denoted by *Root*. Whenever a new tree search is to be started, the root state with the smallest maximum search depth is selected as start state. If there are two or more root states with the same minimum search depth, a root state with minimum path length is chosen among them. After tree search, the maximum search depth is increased by 1.

Without further precautions, the number of root states grows monotonously. On the other hand, there are cases where a root state $r \in Root$ can be dropped or ignored after LTS^{n}_{r} is explored:

1. $Front(LTS^{n}_{r}) = \emptyset$, i.e., no new states have been found (possibly due to a deadlock).
2. $Front(LTS^{n}_{r}) \subseteq Root$, i.e., all newly found states have become root states.

In the first case, an m-bounded DFS ($m > n$) will not lead to any new state. In the second case, all states in $Front(LTS^{m}_{r})$, $m > n$, are also included in $\bigcup_{r' \in Front(LTS^{n}_{r})} Front(LTS^{m-n}_{r'}) \subseteq \bigcup_{r' \in Root} Front(LTS^{m-n}_{r'})$.

For test generation purposes, the root states are of particular interest due to the fact that the execution of their corresponding path results in coverage increase. Therefore, even if any of the two situation above arises, the root state should be kept. On the other hand, if path p_1 of some root state r_1 is a subpath of path p_2 of another root state r_2,

then r_1 can obviously be ignored for test generation. As a consequence, the status of a root state is among the following alternatives:

1. *ok*: The root state is still in use for state space exploration and test generation.
2. *ignore*: The root state is still used for state space exploration but its path can be ignored for test generation.
3. *done*: The root state (actually its path) is stored solely for test generation purposes.

8.3.2 Algorithmic Description

In figures 8.1 and 8.2, the Tree Walk algorithm is given in an abstract programming language. It is divided into two parts: The main function (figure 8.1) deals with the selection of root states, the invocation of the search procedure, and the evaluation of search results. The actual state space exploration is performed by function treeSearch (figure 8.2).

Requirements on the Simulation Environment. The algorithm makes only few requirements on the simulation environment.

For any state $s \in States$ and any event $e \in Events$

- s.nextEvents() returns the set of all events that are executable in state s.
- s.nextState(e) returns the system state s' that is obtained by executing e in s.

Furthermore,

- initialState() returns the initial state of the system.

For each state s, the simulator must store the sequence of events that were executed to reach s from the initial system state (typically realized in terms of a transition stack). This sequence of events can be obtained by calling s.Path(). The other way round, p.endState() returns the state that is reached if path p is explored.

Function treeWalk. The Tree Walk algorithm (algorithm 8.1) starts by initializing the global data structure in which information on the current coverage are stored (line 2). (The hash table operations in lines 3 and 10 are only needed for the extension described in section 8.4). Then, the set of root states is defined (line 4). Initially, there is only one root state with the path being the current path, $depth = 1$, and $status = ok$.

As long as there is any root state with $status \neq done$ and there is no reason to stop Tree Walk, the statements in lines 6 to 25 are executed. Among all root states with $status \in \{ok, ignore\}$, a candidate with minimum depth and path length is chosen (lines 6–8). Then, the system is driven into the end state of the path, i.e., into the root state, and a tree search is invoked (lines 9 and 11).

Besides a revised (enlarged) set of root states, function treeSearch returns two boolean values, named *newState* and *newRoot*. For a tree search with maximum search depth

Algorithm 8.1 TREE WALK – Main function treeWalk

```
 1:  ℙ(Path) treeWalk( maxTime : Time, targetCoverage : ℝ⁺ ) {
 2:    system.resetCoverageTable();
 3:    system.resetHashTable();  // for detection of identical states only
 4:    roots := { ( path:system.initialState().path(), depth:1, status:ok ) };
 5:    while ( ( ∃r ∈ roots : r.status ≠ done ) ∧ continue( maxTime, targetCoverage, ... ) ) {
 6:      currentRoot := choose r ∈ roots : r.status ≠ done ∧
 7:                                ∀s ∈ roots : ( r.depth < s.depth ) ∨
 8:                                    ( r.depth = s.depth ∧ | r.path | ≤ | s.path | );
 9:      state := currentRoot.path.endState();
10:      system.hashTableHit( state, currentRoot.depth + 2 );  // extended version only
11:      ( roots, newState, newRoot ) := treeSearch( roots, state, 0, currentRoot.depth,
12:                                                  maxTime, targetCoverage );
13:      switch ( newState, newRoot ) {
14:        case ( false, false ) :  // no new state found
15:          roots := roots \ currentRoot ∪ { ( path:currentRoot.path,
16:                                depth:currentRoot.depth, status:done ) };
17:        case ( true, false ) :  // new state(s) found, none of them being a new root
18:          roots := roots \ currentRoot ∪ { ( path:currentRoot.path,
19:                                depth:currentRoot.depth + 1, status:currentRoot.status ) };
20:        case ( false, true ) :  // new state(s) found, all of them being new roots
21:          roots := roots \ currentRoot;
22:        case ( true, true ) :  // new state(s) found, some of them being new roots
23:          roots := roots \ currentRoot ∪ { ( path:currentRoot.path,
24:                                depth:currentRoot.depth + 1, status:ignore ) };
25:      }
26:    }
27:    return { p | ∃r ∈ roots :  p = r.path ∧ r.status ≠ ignore ∧ r.path.length() > 0 }
28:  }
```

maxDepth, *newState* is true if any new state has been found at depth *maxDepth* that has *not* become a root state ($Front(LTS^{maxdepth}_{currentRoot}) \setminus roots \neq \emptyset$). Contrarily, *newRoot* = *true* indicates that at least one root state was added at depth *maxDepth*. Depending on the combination of both boolean values, the set of root states is modified as follows:

- If no new state was found at all, the status of the current root state is set to *done* (lines 14–16).
- If only non-root states were found, the maximum search depth of the current root state is increased by 1 (lines 17–19).
- If only root states were found, the current root state is removed from the set of root states (lines 20–21).
- If both root states and non-root states were found, the maximum search depth of the current root state is increased by 1 and its status is set to *ignore*.

The result of the Tree Walk algorithm is the set of non-empty paths which belong to root states with *status* ≠ *ignore*.

Function treeSearch. Function treeSearch (algorithm 8.2) is defined recursively. After checking the abort criteria (lines 3–5), the local variables *newState* and *newRoot* are initialized (line 6). Then all possible events in the current state are evaluated in a backtracking manner (loop from line 7 to 31).

For each event, *nextState* is computed (line 8) and the (preliminary) decision whether *nextState* shall become a root state is stored in the boolean variable *addRoot* (9).

Then, depending on the current depth and the result of function searchDeeperExceptionally (see below), treeSearch is invoked recursively (lines 11 and 12). Since *newState* and *newRoot* keep track whether any new (root) state has been found in the whole subtree, their values are combined with the return values *newState2* and *newRoot2* (line 13).

Depending on the value of *addRoot*, two cases must be considered (lines 17–30): If *nextState* was supposed to become a root state, i.e., *addRoot* = *true*, than *newRoot* is set to *true*. (If *newRoot2* is *true* than *newRoot* has already been *true* before). Next, the return values of the recursive call of treeSearch are examined (compare with the evaluation of *newState* and *newRoot* in function treeWalk):

- If no root state was found, *nextState* is added to the set of root states with *status* = *ok* (lines 19–21).
- If both root states and non-root states were found, *nextState* is added to the set of root states with *status* = *ignore* (lines 19–21).
- If only root states are found, *nextState* is not added to the set of root states.

If *addRoot* is false, the current search depth is compared with *maxDepth* (line 27). If it is greater or equal, *newState* is set to *true* (line 28). New states in a depth lower than *maxDepth* do not modify the variable, because they have been visited before by a shallower tree search.

External Functions. The Tree Walk algorithm can be adapted to specific needs by means of external functions. The concrete implementation of the following boolean functions is left unspecified in the scope of Tree Walk:

- continue(*maxTime*, *targetCoverage*, ...)

  Decides whether Tree Walk shall terminate or continue. For example, Tree Walk might be stopped if the computation time exceeds a user-defined limit or the targeted coverage is reached. Other possible criteria include the current search depth or the number of states visited so far. In a real simulation system, the user should also be able to stop Tree Walk at any time, e.g., by pressing a *cancel* button at the user interface.

Algorithm 8.2 TREE WALK – Subfunction treeSearch

```
 1:  ℙ(Root)×bool×bool treeSearch( roots : ℙ(Root), state : State, depth : ℕ0,
 2:                                maxDepth : ℕ, maxTime : Time, targetCoverage : ℝ+ ) {
 3:    if ( ¬continue( maxTime, targetCoverage, ... ) ) {
 4:      return ( roots, false, false );
 5:    }
 6:    ( newState, newRoot ) := ( false, false );
 7:    for all event ∈ state.nextEvents() {
 8:      nextState := state.nextState( event );
 9:      addRoot := system.increasedCoverage() ∨ addRootExceptionally( ... );
10:      if ( depth + 1 < maxDepth ∨ searchDeeperExceptionally( ... ) ) {
11:        ( roots, newState2, newRoot2 ) := treeSearch( roots, nextState, depth + 1,
12:                                                      maxDepth, maxTime, targetCoverage );
13:        ( newState, newRoot ) := ( newState ∨ newState2, newRoot ∨ newRoot2 );
14:      } else {
15:        ( newState2, newRoot2 ) := ( false, false );
16:      }
17:      if ( addRoot ) {
18:        newRoot := true;
19:        if ( ¬newRoot2 ) {
20:          roots := roots ∪ { ( path:nextState.path(), depth:1, status:ok ) };
21:        }
22:        if ( newRoot2 ∧ newState2 ) {
23:          roots := roots ∪ { ( path:nextState.path(), depth:1, status:ignore ) };
24:        }
25:        // ( newRoot2 ∧ ¬newState2 ) ⇒ all successor states are roots
26:      } else {
27:        if ( depth + 1 ≥ maxDepth ) {
28:          newState := true;
29:        }
30:      }
31:    }
32:    return ( roots, newState, newRoot );
33:  }
```

- increasedCoverage()

  Determines whether the execution of the last event or the reaching of the current system state has resulted in a higher coverage. The exact coverage criteria is left unspecified; in principle, the function can be based on any of the criteria presented in section 4.2.2 on page 53.

- addRootExceptionally()

  Allows to declare a state to be a root state, even if coverage has not increased. An exceptional case in which this might be desirable is the preceding declaration of a root state in which the system was not quiet. If a coverage increase occurs in the middle of a stimulus-response observation step, it makes sense to declare all states as root states up the ones in which the system must wait for new input. In the end, only those states in which the system is quiet will remain in the set of root states while all states with subpaths will be dropped automatically in succeeding steps. In this way, all test sequences generated by Tree Walk are "*complete*" in the sense that they drive the system in a stable state.

- searchDeeperExceptionally()

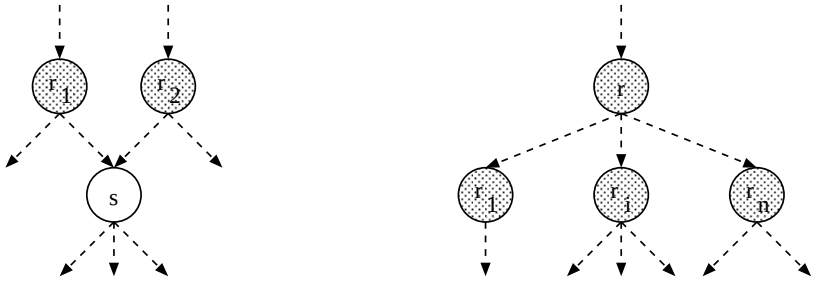
  Allows to further investigate the current path, even if the maximum search depth (*maxDepth*) is reached. E.g., if a coverage increase is detected at *maxDepth*, it is reasonable to continue the current path, since it is likely (according to the Tree Walk assumption) to find another root state at *maxDepth + 1*. This way, a lot of shallow tree searches and the introduction of many temporary root states can be avoided.

8.4 Detection of Identical States

In order to cope with the state space explosion problem, the Tree Walk algorithm should be combined with state space reduction techniques and heuristics. One of the most efficient techniques is to prune the current search path if a state is reached that has been visited before. But for that purpose, all states must be stored in memory which is practically impossible, even if the states are stored in compressed form.

Therefore, the Supertrace (bit-state) algorithm (Holzmann, 1991) is used in practice. It requires far less memory but might prune paths erroneously. The underlying idea is to compute a hash key for each state and to mark the visit of the state by a single bit in the corresponding entry in a hash table. To reduce the risk of undetected clashes (two different states have the same hash key and thus are considered identical), a second hash function can be used such that each state maps to two bits in the hash table. In practice, the Supertrace algorithm has gained great popularity due to its simple implementability.

Unfortunately, the Supertrace algorithm cannot be applied directly to Tree Walk, because Tree Walk requires explicitly that a lot of states are revisited. There are two reasons: First, several explorations with increasing maximum search depth are started at each root state. For a k-bounded search, all states up to depth $k-1$ have been visited



(a) Two root states with common substate (b) A root state with root states along all paths

Figure 8.1: Tree Walk – Detection of identical states

in former passes. Second, the root state changes after almost every exploration. If some state s is reachable from two root states r_1 and r_2, the Supertrace algorithm should not be applied unrestrictedly, because it might stop tree searches unintentionally. In particular, if $r_1 \xrightarrow{e} r_2$, a tree search starting at r_1 with depth 2 marks all successor states of r_2 as visited such that a later exploration starting at r_2 would terminate immediately.

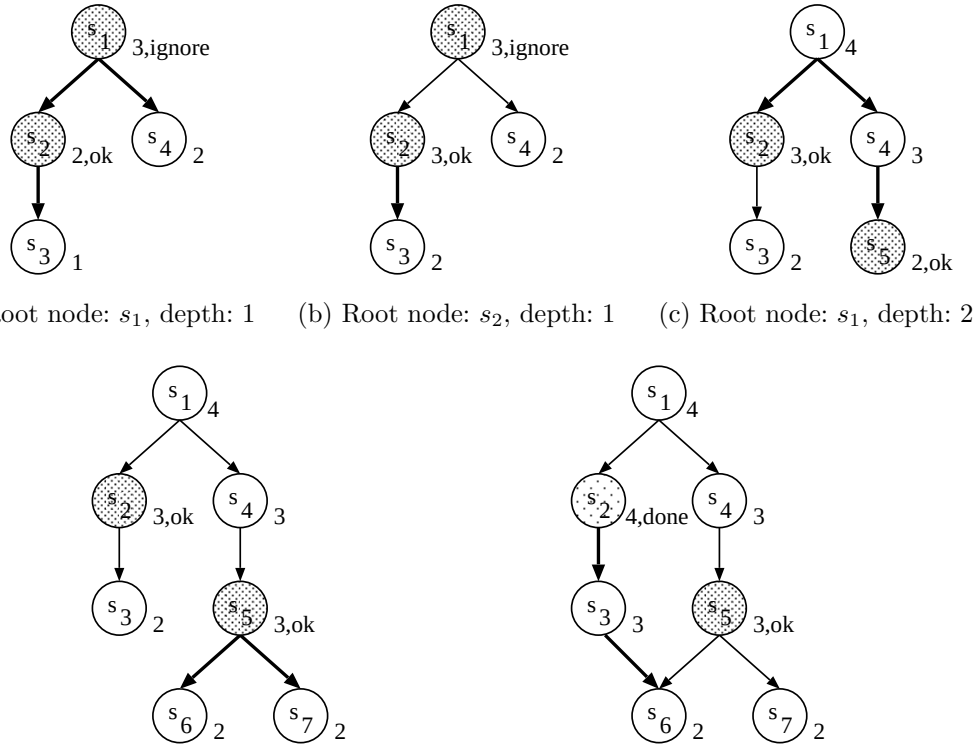
In a simple approach, the hash table used for the Supertrace algorithm could be cleared after each single tree search. But this method is not optimal as useful information about previously visited states is lost. In figure 8.1, two scenarios are presented where a *global* solution is beneficial.[2]

In figure 8.1(a), two root states r_1 and r_2 have a common successor state s_3, i.e., $r_1 \rightarrow^* s_3$ and $r_2 \rightarrow^* s_3$. It is assumed that there is no root state on all paths from r_1 to s_3 and r_2 to s_3. Obviously, the state space spanned by s_3 does not have to be explored twice. But this raises the question whether it should be investigated during the tree search started at r_1 or at r_2. A k-bounded DFS started at r_1 corresponds to a $(k - |r_1, s_3|)$-bounded DFS for state s_3. If it turns out, during a successive l-bounded DFS at r_2, that $l - |r_2, s_3| > k - |r_1, s_3|$, then the exploration of LTS_{s_3} should be repeated for maximum penetration. On the other hand, a $k+1$-bounded DFS at r_1 should not consider LTS_{s_3} again, since $k + 1 - |r_1, s_3| \leq l - |r_2, s_3|$.

Figure 8.1(b), illustrates the situation where each path from root state r will eventually lead to another root state r_i. Obviously, the simulation of LTS_r can be stopped in each root state r_i, since a separate exploration is started for LTS_{r_i}. Moreover, state r should be removed from the list of root states as soon as no new state can be found. Let k be the maximum of $\{|r, r_i| \,|\, i \in \{1 \ldots n\}\}$. Then no further tree search has to be initiated for r after a k-bounded DFS.

The examples above demonstrate that it is not sufficient to mark the visit of a state by a single bit in a hash table. Instead, some additional information about *how* the state has been visited must be stored. The fundamental idea is to store the *lookahead*

[2]To simplify the representation of identical states, the state space is visualized by a graph instead of a tree in the following figures. This approach is valid because each behavior tree can be transformed into an equivalent reachability graph and vice versa.



(a) Root node: s_1, depth: 1 (b) Root node: s_2, depth: 1 (c) Root node: s_1, depth: 2

(d) Root node: s_5, depth: 1 (e) Root node: s_2, depth: 2

Figure 8.2: Running Tree Walk with detection of identical states

of a state, i.e., how deep the state space has been explored with the current state as origin. If a k-bounded DFS is made at root state r, then r has lookahead $la = k + 2$. A state s that is visited with path p starting at the root state and $|p| = n$ has lookahead $la = max(k + 2 - n, 1)$.

Whenever a new state s is investigated, a hash key is computed and the corresponding hash table entry is looked up. If the current lookahead la is lower than or equal to the value stored in the hash table, then LTS_s^{la} has already been explored and thus the exploration can be stopped. Otherwise, la is registered in the hash table. If more than one hash key is computed for each state, the hash table entries must be checked for equality. In case they are different, a clash is detected and the state space exploration continues without altering the hash table.

For the computation of lookahead la, an offset of 2 is used. There are two reasons why this offset is needed: First, the lookahead should always be ≥ 1, because 0 indicates an empty field in the hash table. Therefore, an offset of 1 is needed for states found at the maximum search depth. Moreover, the Tree Walk algorithm is designed to explore states even beyond the maximum search depth k in exceptional cases. In order to be able to distinguish between states at depth k and $k' > k$, the offset has to be set to 2. The following example points out why such a distinction is important.

8.4.1 Example

The functionality and efficiency of the lookahead mechanism is illustrated by the example in figure 8.2. It demonstrates how TREE WALK explores the state space of a system of which only the start state s_1 is known initially. The states are numbered in the order of occurrence. Root states are represented as filled circles. Next to each state, its status and the lookahead stored in the hash table are printed.

First, TREE WALK puts s_1 on the list of root states and starts a DFS bounded to depth 1. It turns out that s_1 has two successor states, s_2 and s_4 and that either the event of the transition from s_1 to s_2 or the state s_2 itself causes a coverage increase. As a consequence, the state space exploration is continued exceptionally in root state s_2 which leads to the detection of state s_3. Thereafter, s_2 becomes another root state and the status of s_1 is set to *ignore*. During simulation, the lookahead is computed for each state and registered in the hash table. Figure 8.2(a) depicts the situation after the DFS is completed.

In a second step, a DFS with maximum depth 1 is started at root state s_2. This time, the lookahead for s_3 is 2, i.e., greater than in the previous run, and thus the hash table entry is updated. Equally, the lookahead for s_2 is increased to 3 (see figure 8.2(b)). No new states are discovered during this search.

Since there is no other state on the root state list, another exploration is started at state s_1 but this time with 2 as maximum search depth. For s_2, the current lookahead is $2 + 2 - 1 = 3$ which is equal to the lookahead stored in the hash table. This means that the exploration can be stopped at this state because all states that are reachable, i.e., all state in $LTS^1_{s_2}$, have already been investigated before. Independently from this, a new state s_5 is found on the other branch. Due to increased coverage, s_5 becomes a new root state. The only state in $LTS^2_{s_1}$that was not already included in $LTS^1_{s_1}$ is a root state. Therefore, s_1 can be removed from the list of root states (see figure 8.2(c)).

Next, a tree search at root state s_5 is started. Two new states, s_6 and s_7, are found and their lookaheads (2) are stored in the hash table (figure 8.2(d)).

Again, a new DFS with maximum depth 2 is started at s_2. The only successor state of s_3 is s_6. According to the formula presented in the previous section, the current lookahead for s_6 is $2 + 2 - 2 = 2$. Since the lookahead for s_6 in the hash table is 2 as well, the exploration can be stopped. Since no new state is found (neither root state nor common state), the status of s_2 is set to *done*, i.e., it will not be considered for further explorations. But in contrast to s_1, s_2 is not removed from the root state list because there is no root state whose path subsumes the path of s_2.

8.4.2 Algorithmic Description

An revision of algorithm 8.2 that supports the detection of identical states is given as algorithm 8.3. In comparison to the original version, lines 10, 11, 19, 22, and 30–32 have been added and the condition in line 34 has been modified.

Algorithm 8.3 Tree Walk – Subfunction treeSearch with detection of identical states

```
 1: ℙ(Root)×bool×bool treeSearch( roots : ℙ(Root), state : State, depth : ℕ0,
 2:                               maxDepth : ℕ, maxTime : Time, targetCoverage : ℝ+ ) {
 3:   if ( ¬continue( maxTime, targetCoverage, ... ) ) {
 4:     return ( roots, false, false );
 5:   }
 6:   ( newState, newRoot ) := ( false, false );
 7:   for all event ∈ state.nextEvents() {
 8:     nextState := state.nextState( event );
 9:     addRoot := system.increasedCoverage() ∨ addRootExceptionally( ... );
10:     tableHit := system.hashTableHit( nextState, max( maxDepth - depth + 1, 1 ) );
11:     if ( ¬tableHit ) {
12:       if ( depth + 1 < maxDepth ∨ searchDeeperExceptionally( ... ) ) {
13:         ( roots, newState2, newRoot2 ) := treeSearch( roots, nextState, depth + 1,
14:                                                       maxDepth, maxTime, targetCoverage );
15:         ( newState, newRoot ) := ( newState ∨ newState2, newRoot ∨ newRoot2 );
16:       } else {
17:         ( newState2, newRoot2 ) := ( false, false );
18:       }
19:     }
20:     if ( addRoot ) {
21:       newRoot := true;
22:       if ( ¬tableHit ) {
23:         if ( ¬newRoot2 ) {
24:           roots := roots ∪ { ( path:nextState.path(), depth:1, status:ok ) };
25:         }
26:         if ( newRoot2 ∧ newState2 ) {
27:           roots := roots ∪ { ( path:nextState.path(), depth:1, status:ignore ) };
28:         }
29:         // ( newRoot2 ∧ ¬newState2 ) ⇒ all successor states are roots
30:       } else {
31:         roots := roots ∪ { ( path:nextState.path(), depth:1, status:done ) };
32:       }
33:     } else {
34:       if ( ¬tableHit ∧ depth + 1 ≥ maxDepth ) {
35:         newState := true;
36:       }
37:     }
38:   }
39:   return ( roots, newState, newRoot );
40: }
```

Algorithm 8.4 TREE WALK – Hash table access with n keys

```
1:  bool hashTableHit( state : State, lookahead : ℕ ) {
2:    (hkey_1,hkey_2, ..., hkey_n) := state.hashKeys();
3:    if ( ∀i,j ∈ {1...n} : htable[ hkey_i ] = htable[ hkey_j ] ) {
4:      if ( htable[ hkey_1 ] ≥ lookahead ) {
5:        return true;
6:      } else {
7:        for all i ∈ {1...n} {
8:          htable[ hkey_i ] := lookahead;
9:        }
10:       return false;
11:     }
12:   } else {
13:     return false;
14:   }
15: }
```

Whenever a new state is reached, TREE WALK checks whether this state has been visited before with a larger lookahead. This is achieved by calling function hashTableHit in line 10. If its return value (assigned to variable *tableHit*) is *false*, i.e., the state must be investigated, algorithm 8.3 behaves identical to algorithm 8.2. Otherwise, the current path is pruned. Even if a state is revisited and *tableHit* = *true*, it may nevertheless be identified as a root state, because the transition leading to the state results in increased coverage. In this case, the state is added to the set of root states with status *done*.

The definition of hashTableHit is given as algorithm 8.4. hashTableHit first calls some function hashKeys which returns n hash keys for a given state (line 2). Then, the hash key entries in the global hash table *htable* are checked for equivalence (line 3). If their values are different, they must spring from at least two states that are different from the current state. That means there is a (detectable) clash in the hash table and hashTableHit returns *false* (line 13). Otherwise, the *lookahead* of the current state is compared with the hash table entries (line 4). If it is lower or equal, hashTableHit returns *true* (line 5). If the lookahead is greater than the current hash table entries, the hash table is updated accordingly and *false* is returned (lines 7–10).

The presented algorithm does not make any assumption on the number of bits used for storing the lookaheads in the hash table. For simple implementation, a slot size of 8 bits might be chosen. It allows for tree searches with a maximum search depth of 253. However, the case studies described in section 8.5 indicate that the maximum search depth is typically much smaller (4 and 17 resp.). Thus, the slot size can be reduced to 5 or 6 bits which increases the number of states that can be registered in memory by 37.5% or 25%.

8.5 Case Studies

For proving the superiority of Tree Walk over Random Walk and iterative depth-first search, all three search strategies have been applied to the SDL systems of the Inres and VB5.2 protocol. The aim was to explore the state spaces of both protocols and to obtain the highest possible symbol coverage (i.e., C0 coverage) while exploring a minimum number of states.

The tests were performed with the Validator of the Telelogic Tau 4.2 tool suite which supports all three exploration techniques.[3] The state space exploration was stopped when a symbol coverage of 100% was achieved or the computation time exceeded 1 hour. All tests were performed on a SUN Sparc Ultra II with two 300 Mhz processors (of which only one was used by the Validator) and the Solaris 7 operating system.

For Random Walk, the search depth was restricted to 100 transitions and the number of repetitions was set to a very large number to prevent premature termination. For iterative DFS, the initial maximum search depth was set to 5 and incremented by 5 with each iteration. For comparable results, the iterative DFS was combined with the bit-state algorithm. For the Inres protocol, a hash table with 8,000,000 entries (bits) was used, for VB5.2 the hash table size was set to 240,000,000 bit fields. The Tree Walk implementation did not require any customization. Its hash table is enlarged automatically by a factor of 2 if it is filled by more than 3 percent, starting with an initial size of 1,048,575 bytes.

The choice of appropriate heuristics as well as the set of possible input signals have a strong influence of the test results. In both case studies, the default settings of the Tau Validator for state space exploration were kept since (a) the settings restrict the state space without lowering the attainable symbol coverage for most protocols and (b) a common user isn't likely to modify them.

Among others, the default settings assume that SDL transitions are non-interruptible and that the time it takes for the SDL system to perform internal actions is very small compared to timeout values and the response time of the environment. Moreover, only the first process instance in the ready queue is allowed to execute and the input port queue of each process is restricted to three signals. If a violation of the dynamic semantics of SDL takes place (e.g., by an implicit signal consumption), the exploration is stopped at the current state.

The Inres Case Study. For the exploration of the state space of the Inres protocol, the Validator's default set of input signals has been extended in such a way that a symbol coverage of 100% could be achieved.[4] Actually, total symbol coverage was gained

[3] Tree Walk has been implemented by the author and is an integral part of the tool suite.

[4] The precise set of signals comprises *ICONreq*, *IDATreq(0)*, *IDISreq*, *MDATreq((.CR, zero, 0.))*, *MDATreq((.CC, zero, 0.))*, *MDATreq((.AK, zero, 0.))*, *MDATreq((.AK, one, 0.))*, *MDATreq((.DT, zero, 0.))*, and *MDATreq((.DR, zero, 0.))*, where *MDATreq((.CR, zero, 0.))* and *MDATreq((.DT, zero, 0.))* are syntactically correct but, of course, semantically invalid signals.

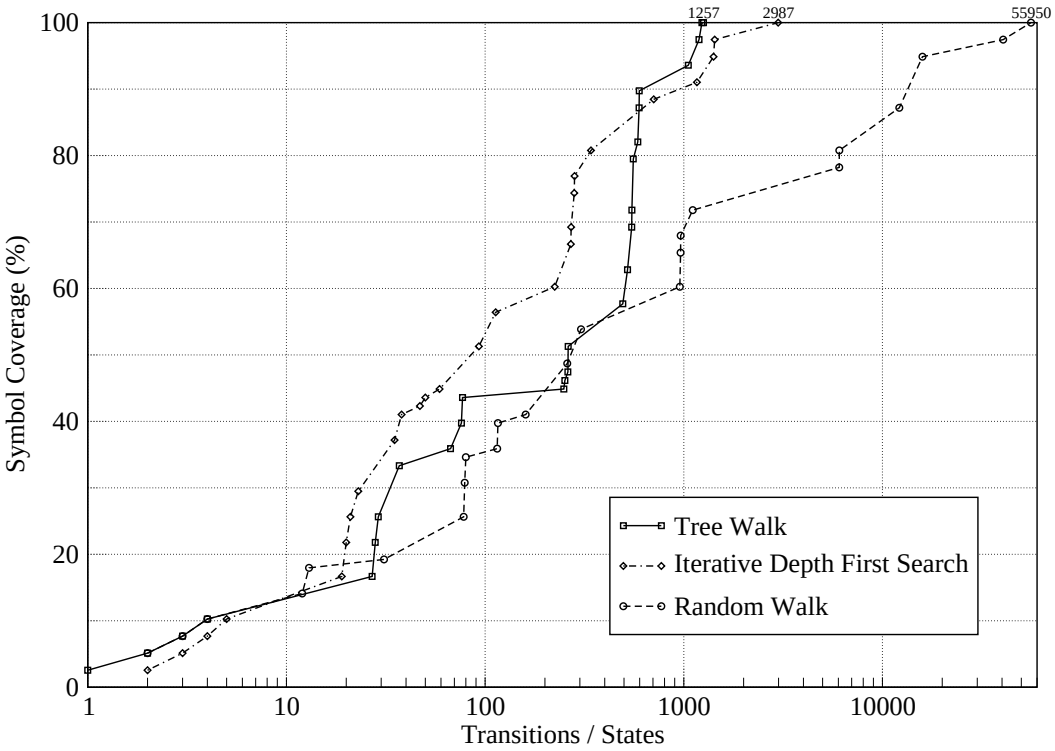


Figure 8.3: Exploration of the Inres protocol

with Tree Walk, Random Walk, and iterative DFS within 1 to 12 seconds.

Figure 8.3 shows the growth of symbol coverage versus the number of computed transitions/states (states that were reached more than once, were counted multiple times). One curve is given for each search strategy. The diagram indicates that Tree Walk outperforms the two other search strategies by only having to compute 1257 states to reach 100% symbol coverage. Iterative DFS must explore more than twice the number of states (2987) and Random Walk lags far behind with 55950 states.

During state space exploration, Tree Walk adds 62 states to the list of root states. 32 of them are removed again from the list later when they are proven not to lead to any new state. The maximum path length (measured from the initial start state) is 32 transitions. In total, 124 tree searches are initiated from the 62 root states. The maximum search depth is 4 transitions, i.e., only very shallow explorations are needed.

Tree Walk produces 9 traces whose execution causes 100% symbol coverage. In figure 8.5, these traces are shown as MSC diagrams. Apparently, the traces look very similar to traces produced manually by a test specifier. For example, *InresTreeWalk_7* and *InresTreeWalk_8* correspond to test cases where the tester does not respond to a request at PCO *MSAP2*. In *InresTreeWalk_9*, the responder acknowledges the data transfer (DT) with the wrong sequence number. As a consequence, the data packet is re-sent (and ignored three times in the following). All traces describe sensible and coherent scenarios — there is no way to simplify them.

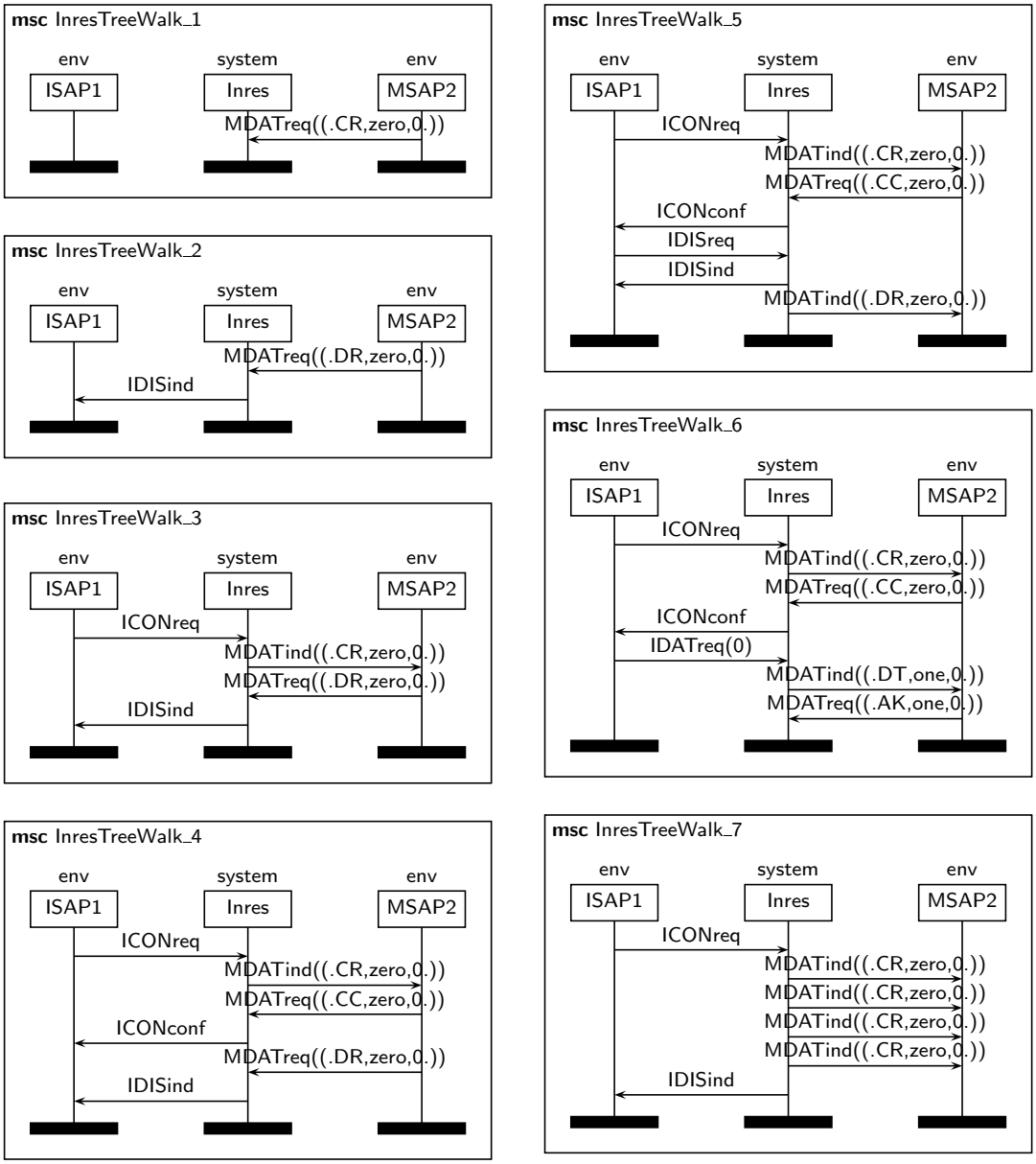


Figure 8.4: MSCs generated by Tree Walk

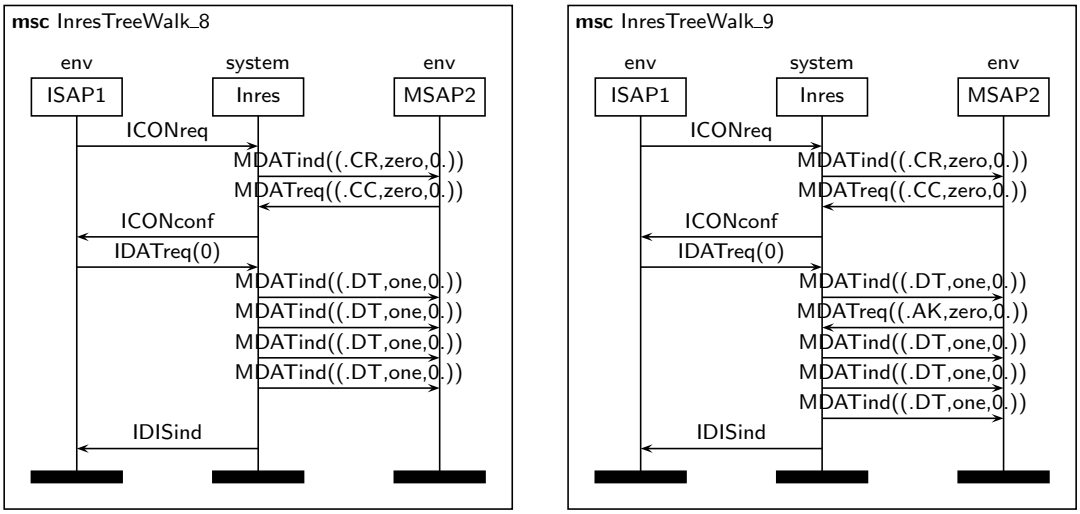


Figure 8.5: MSCs generated by Tree Walk (continued)

The VB5.2 Case Study. The VB5.2 protocol requires very complex signal parameters. Since the Tau Validator was not able to define a reasonable default set of SDL signals for stimulating the system, 48 signal definitions were taken from MSC test purposes defined within ETSI project STF 151.

Even though the extracted signals are supposed to be sufficient for the most common traces, none of the three search strategies was able to achieve a symbol coverage of at least 33% within one hour. One explanation for this fact is that the SDL specification is triggered by several external synonyms which, e.g., activate some special error insertion code. For that reason, some portions of the SDL system are not reachable in normal case.

In figure 8.6, the symbol coverage is shown for each search strategy. Due to the large number of computed transitions/states, a logarithmic scale is chosen for the horizontal axis. In analogy to the Inres case study, Tree Walk once again proves to be superior to the other two search strategies. It takes Tree Walk 75,704 states to achieve 32.51% symbol coverage while Random Walk must visit 972,528 states for a symbol coverage of only 30.88%. The application of iterative DFS to the VB5.2 protocol results in poor coverage (27.70%), since a maximum search depth of only 55 transitions was possible within one hour.

Tree Walk examines the state space starting from 417 different root states. The longest path between the initial system state and a root state comprises 78 transitions. 1561 tree searches are started from these root states. The maximum search depth is 17 but only 51 out of the 417 root states are used for explorations with a maximum search depth ≥ 4. Due to the efficient detection of identical and redundant states, 370 root states can be dropped prematurely. Based on the remaining 47 root states, Tree Walk outputs 33 traces; the paths of 14 root states are ignored because they are contained as prefixes in other paths.

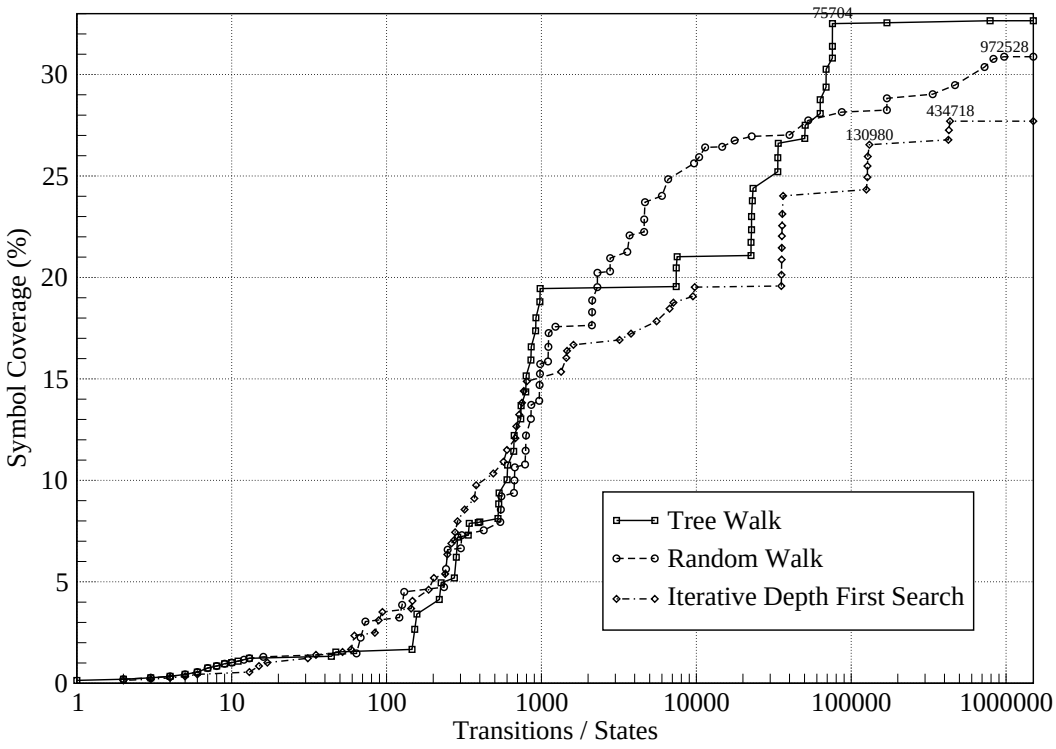


Figure 8.6: Exploration of the VB5.2 protocol

8.6 Discussion

The Tree Walk algorithm has been implemented in Autolink. Its efficiency has been demonstrated by the two case studies described in the previous section.

A major advantage of Tree Walk from the user's point of view is that it does not need to be configured. Tree Walk only requires a termination criteria, i.e., a target coverage and an expiration time, to do its job. In contrast, the iterative DFS strategy requires a maximum search depth and – if used in conjunction with the Supertrace algorithm – a fixed hash table size. This information must be provided by the test specifier who is often not able to assess the impact of the options on the outcome and who needs several trials until an optimal result is obtained.

Despite the benefits of Tree Walk there are a few aspects that must be taken in account when running Tree Walk. There are also possible further improvements that are sketched in the following.

Assumption on Quiescence. An important prerequisite for the use of Tree Walk (at least as implemented in Autolink) is that, starting from *any* state, the simulated system eventually becomes quiescent if no further input is made. Whenever coverage is increased, Tree Walk continues the state space exploration as long as the system is

able to perform some action – regardless of the maximum search depth. As mentioned before, this extension is useful to produce test cases that include all responses to a former stimulus, i.e., that do not end abruptly with a signal sent from the test environment.[5] Unfortunately, there are specifications that violate this prerequisite. However, such undesired behavior can often be suppressed by filtering heuristics such as assigning low priorities to timer expiration events and SDL spontaneous transitions.

Exceptional Behavior. Without provisions, Tree Walk — like any other search strategy — produces both traces that describe regular behavior as well as traces that describe exceptional behavior. The latter traces are not always wanted as test purposes.

For instance, consider the medium process as part of the Inres SDL system. In the original specification, the medium may transmit or drop signals. During the test purpose specification phase, the indeterministic choice and the latter alternative should be removed from the specification. In other cases, filtering heuristics may allow to hide unwanted behavior such that the specification itself does not have to be modified.

Missing Postambles. The purpose of Tree Walk is to find paths whose execution results in a high coverage — it is not concerned with finding postambles. Thus, the paths obtained by Tree Walk should be completed in such a way that the system is driven in its initial state again. Preferably, a breath-first search or iterative DFS should be used for that purpose.

Computation of the Effective Coverage. When generating a test case from a path generated by Tree Walk, one abstracts from all internal behavior. Thus, it is possible that a non-deterministic system executes different internal actions while showing the same input-output behavior. Just like the SaMsTaG tool proves that a *possible pass observable* (PPO) is a *unique pass observable* (UPO), the traces generated by Tree Walk might be checked in an additional step that they always increase coverage.

[5] In addition, the exploration will likely result in further coverage increases such that it is reasonable to proceed the current exploration.

9 Test Suite Representation

One aspect of automatic test generation that is neglected in literature is the appropriate representation of test suites. Tests derived automatically from a formal specification look different from test suites written by hand. Tools produce test suite documents in a uniform manner based on templates. Test suites developed manually do not have the same degree of uniformity as they evolve step-by-step. But, on the other hand, they tend to be more compact and more readable. There are two major reasons why test generation tools fail with regard to the readability of their test suites:

- *Generic names* are assigned to entities that cannot be associated with named entities in the formal specification.
- *Flat structures* are used for the representation of both data and behavior.

The problem of generic names concerns all kinds of entities: Test cases and test steps, constraints, formal parameters, etc. Readability requires intuitive names that indicate what an entity stands for. For example, the name of a test case should clearly describe its test purpose; a constraint name should indicate which part of the data definition is most important.

The lack of structuring results in large test suites in which a lot of information is replicated. For reusability purposes, test cases should be structured into several test steps. This also makes the inter-relations and intra-relations of test cases explicit in a test suite document. Similarly, constraints of the same type often only differ in just a few data fields and should be combined by means of parameterization, derivation, and chaining to provide a compact representation.

The problems described above can be reduced to the same cause, namely the lack of information about the *meaning* and *relationship* of entities. What is tested by a specific test case? What is different/common between two similar constraints of which an appropriate structuring can be deduced?

There are different ways to overcome the problem:

1. User defined rules

   The test specifier tells the test generation tool explicitly how to structure a test suite. This approach is adequate if test generation proceeds in an interactive way. For example, in the Autolink tool, the user can specify test purposes as message sequence charts whose structure is preserved during the mapping to TTCN. Moreover, Autolink's configuration language can be used to name and structure TTCN constraints and arrange test cases and test steps in test groups.

2. Syntax-directed heuristics

   The specification on which automatic test generation is based may also give some hints on the meaning of a test. For example, if an SDL *save* construct is executed during simulation, the test might be classified as a behavior test rather than a basic interconnection test, because some exceptional case is covered.

3. Information derived during test case generation

   The test case generation process itself may suggest a reasonable structuring and naming. For instance, information about which part of a generated trace refers to the test purpose can be derived. If test generation is based on transition coverage, the input signal which triggered the covered transition, the process in which the transition took place, and the start and end state of the transition can be used as criteria for naming and structuring test cases. This approach has been used in the TestComposer tool.

4. Heuristics based on statistical analysis

   If the same information reoccurs at several places in a document, it is of advantage to merge them. Moreover, if there is a set of entities of the same type and one of these entities is referred to more often than all others, the conclusion may be drawn that this entity plays the role of a default or denotes the "normal" case.

Even though one may argue that an automatically generated test suite is not intended primarily for human inspection, there are good reasons to improve their readability: Test tools generate abstract test suites (ATSs). These ATSs have to be transformed further into *executable* test suites before they can be applied to a concrete SUT. This processing step requires the intervention of a test realizer who defines a mapping of ATS elements (e.g., PCOs, send/receive events) to the components of a concrete test system. ATSs are also published as standards by organizations such as the ATM Forum or ETSI. And most important, the assessment of (negative) test results becomes easier if an ATS provides precise and well-structured information.

This chapter presents different solutions to overcome the readability problem. In section 9.1, the Autolink script language is presented. It allows to control the appearance of generated test suites by user-defined rules. In particular, the naming and parameterization of TTCN-2 constraints and the structuring of test cases into test groups are considered. Section 9.2 deals with the automatic structuring of constraint descriptions. A pragmatic, stepwise procedure is presented that works in polynomial time. Its effectiveness is demonstrated by two case studies. To simplify the implementation of a prototype for this procedure, a generic language for pattern matching and manipulation of lists has been developed. It is described in section 9.3. Finally, section 9.4 discusses some possible future improvements.

9.1 The Autolink Script Language

Within the Autolink project, a script language has been developed that allows the test specifier to control the appearance of generated test suites. The Autolink script

language addresses two problem areas:

1. The representation of constraints (including the introduction of test suite constants and parameters)
2. The structuring of test cases and test steps into test groups

A script is defined once before a test generation is started. Its automatic evaluation saves a lot of manual post-processing of a generated TTCN test suite. In particular, this holds for the case that the SDL specification needs to be refined and thus the test generation process has to be repeated.

9.1.1 General Language Concepts

An AUTOLINK configuration script consists of a sequence of constraint rules, test suite structure rules, and auxiliary functions. Rules and functions can be mixed arbitrarily in a configuration script. AUTOLINK evaluates them in the order in which they are defined. If several rules are applicable at run-time, only the first one is considered. As a consequence, more specific rules should be put on top of default rules.

One design goal of the script language was simplicity. Even an unexperienced user shall be able to comprehend the meaning of existing scripts and specify his own set of rules within a short period of time. The script syntax is not very strict in the sense that, for example, function parameters do not have to be declared. Instead, semantic inconsistencies are checked and resolved at run-time like in many other script languages.

The only data type available in the script language is *text* (character string). Expressions are constructed by basic elements, called *atoms*. An atom is either a simple text, a pattern, a function call, or some context-dependent operator. A list of all types of atoms is given in figure 9.1. An atom always evaluates to text at run-time. Atoms of different kinds can be concatenated by means of the "+" operator to build complex *terms*.

9.1.2 Constraint Rules

The AUTOLINK script language allows to specify how SDL signals are mapped to TTCN constraints during the test generation process by a set of user-defined constraint rules. In detail, these rules address the following issues:

Naming of constraints Without user assistance, constraint names have to be created generically. For instance, a constraint may be named after its signal or the test case in which it is used. If there are different constraints with the same name, they must be distinguished by an additional sequence number. In practice, such a naming scheme is neither satisfying nor flexible enough. By means of constraint rules, the test specifier is able to define his own mapping.

Parameterization of constraints Parameterization is a proper means to avoid a vast number of similar constraints. Moreover, values that are strongly context-dependent become *actual* parameters and are defined directly within that context, i.e.,

- **Simple texts (e.g., `"Request"` or `Request`)**
  Texts must be enclosed by quotation marks if they do not consist of a letter or underscore followed by an alphanumeric character or underscore.
- **Patterns (e.g., `"Sig*"`)**
  Texts with the special characters "`*`", "`?`", and "`[...]`" are interpreted as patterns in the header of constraint and test suite structure rules, as well as in conditions.
- **Function calls (e.g., `OpName($3)`)**
  Function calls are allowed in the body of constraints rules, test suite structure rules, and functions.
- **References to parameters (e.g., `$0`, `$2`)**
  Within the body of a constraint rule, `$`*n* denotes the value of the *n*-th signal parameter; within a function body, it returns the *n*-th function parameter. As a special case, `$0` denotes the name of the signal or function itself.
- **References to atomic expressions (e.g., `@2`)**
  Within the body of a constraint or test suite structure rule, `@`*n* denotes the value matching the *n*-th atom in the rule header. E.g., if a rule with expression `"a" + "*" + "c"` in the header is applied to input `"abc"`, then `@2` is equal to `"b"`. In rules with alternative terms in the header, `@`*n* refers to the *n*-th atom in the term that actually matches the input.

Figure 9.1: Atomic expressions of the AUTOLINK script language

in the constraint references in the specific dynamic behavior description. This improves the readability.

Replacement of parameter values by wildcards If, during test execution, some parameter value of a received message is irrelevant for the computation of the test result, or the exact value is unpredictable (because it depends on the test history) than the signal parameter should be represented by an expression with a TTCN matching mechanism in the constraint declaration table. However, AUTOLINK demands that all signals have specific parameter values in an MSC test purpose.[1] If the test specifier wants to replace some values by arbitrary expressions afterwards, this can be achieved by a script rule.

Introduction of test suite parameters and constants Test suite *constants* are useful if a concrete parameter value does not give any clues about its meaning and hence should be replaced globally by a more meaningful name. Test suite *parameters* are similar to constants except that no value is specified for them. Test suite parameters should be introduced if signal parameters are implementation dependent. For instance, if a telecommunication system is to be tested that supports free phone

[1] This requirement has both procedural and technical reasons: When generating MSC test purposes, AUTOLINK explores the state space of the SDL specification with a number of preset and concrete input data. As a consequence, the resulting MSC test purposes have signals with concrete data as well. Moreover, the simulator engine which AUTOLINK is based on is not able to check whether a specific value matches an expression with matching mechanism.

```
TRANSLATE TC_ContinueReq
  CONSTRAINT NAME "C_TC_ContinueReq"
             PARS $1="Dialog_ID"
END
```

(a) Constraint rule

ASN.1 ASP Constraint Declaration		
Constraint Name	:	C_TC_ContinueReq_001(Dialog_ID : DialogIDtype)
ASP Type	:	TC_ContinueReq
Derivation Path	:	
Comments	:	
Constraint Value		
{ dialogIDtype1 Dialog_ID, tCoriginType2 oSCF }		
Detailed Comments :		

(b) Constraint Declaration Table

Figure 9.2: AUTOLINK script language – A simple constraint rule

numbers, the phone number prefix should not be hard-coded in the ATS. Instead, it should become a test suite parameter, since it varies with the countries. Within OSI's CTMF, each test suite parameter refers to a PIXIT document.

In the following, the constraint rules of AUTOLINK's script language are described by examples of the INAP CS-2 case study.

Constraint rules can be considered as mapping rules: AUTOLINK translates an SDL signal into a suitable TTCN constraint. Each constraint rule consists of two parts: The header specifies one or more signal types to which the mapping shall apply. The rule body defines how concrete signal instances of these types shall be represented in TTCN-2.

A simple constraint rule is shown in figure 9.2(a). The rule states that signals of type *TC_ContinueReq* shall be mapped to constraints whose name is *C_TC_ContinueReq*. If more than one constraint is built during the test generation, all constraints are distinguished by an additional sequence number. Moreover, AUTOLINK parameterizes the resulting constraints with the first parameter of the corresponding signals (referred to by *$1*). The name of the formal parameter used in the constraint declaration table is *Dialog_ID*. An exemplary constraint declaration table for signal *TC_ContinueReq(51,oSCF)* is shown in figure 9.2(b). The value 51 has become an actual parameter and is specified in the references to this constraint in the dynamic behavior part.

The AUTOLINK script language also allows to define a single constraint rule for more than one signal type. This is especially useful if there are similar signal types which can be treated in the same way. The constraint rule in figure 9.3 is applied if a constraint is created for a signal of either type *CallProgressInd* or *CallProgressReq* during the test generation. Following the notation for parameters, the name of the signal itself can be accessed by *$0*. The value of *$0* (as well as *$1*, *$2*, ...) depends on the concrete signal

```
TRANSLATE CallProgressInd | CallProgressReq
  CONSTRAINT NAME "C_" + $0
             PARS $1="callRef"
END
```

Figure 9.3: AUTOLINK script language – A constraint rule for multiple signal types

translated at run-time. The resulting constraint name consists of the concatenation of text "*C_*" and the name of the actual signal.

In addition, a formal parameter with name *callRef* is introduced for the first signal parameter. When reading a constraint rule, AUTOLINK performs static checks to ensure that all signal types declared in the rule header have at least as many parameters as required in the optional `CONSTRAINT PARS` statement.

Constraint names may not only be composed of plain text and signal names. They can also depend on signal parameters. However, in some cases it is not desirable to take the textual representation of a parameter value directly as part of a constraint name. E.g., a protocol engineer might encode complicated signal information with abbreviations or numbers. But for the TTCN test suite output, these abbreviations should be mapped to extended, more meaningful names.

For that purpose, functions can be defined which take an arbitrary number of input parameters and map them to some text. A function consists of a sequence of condition/term pairs that are evaluated from top to bottom. As soon as a condition evaluates to true, the text described by the corresponding term is returned. A condition checks whether two terms evaluate to the same text. Single term comparisons can be combined by the `AND` operator.[2] Typically, a function parameter is compared successively with predefined values. Function parameters can be accessed by `$n`, i.e., in the same way as signal parameters in a constraint rule (cf. figure 9.1).

In figure 9.4(a), the fourth parameter of *TC_InvokeReq* is taken as input for function *OpName*. Depending on its value (denoted by *$1*), the function returns a text which forms the second half of the constraint name. As a consequence, *TC_InvokeReq* signals whose fourth parameters differ are mapped automatically to constraints with different names. A possible constraint declaration table for a *TC_InvokeReq* signal is shown in figure 9.4(b).

If the translation of a signal should not only depend on the signal type but also on some signal parameter value, conditional rules can be defined by means of `IF` statements in the rule body. Only if the conditions in an `IF` statement are satisfied, the constraint is built according to the subsequent specification. A rule body can have several `IF` clauses and a final unconditioned block. The first clause whose condition evaluates to true (or which has no `IF` statement at all) is applied.

[2] There is no explicit `OR` operator. However, the consecutive evaluation of conditions in functions and rules equals a disjunction.

```
TRANSLATE TC_InvokeReq
  CONSTRAINT NAME "CIR_" + OpName($4)
             PARS $1="Invoke_ID", $2="Dialog_ID"
END

FUNCTION OpName
    $1 == "ASF"   : "ActivateServiceFiltering"
    ...
  | $1 == "RC"    : "ReleaseCall"
    ...
  | $1 == "SL_R"  : "SplitLegResult"
  | TRUE          : "UnknownOpCode"
END
```

(a) Constraint rule and function

ASN.1 ASP Constraint Declaration		
Constraint Name	:	CIR_ReleaseCall(Invoke_ID : InvokeIDtype; Dialog_ID : DialogIDtype)
ASP Type	:	TC_InvokeReq
Derivation Path	:	
Comments	:	
Constraint Value		
{ invokeIDtype1 Invoke_ID, dialogIDtype2 Dialog_ID, opClassType3 4, opCodeType4 RC, timeoutValType5 short, argType6 rCArg : initialCallSegment : PIX_ReleaseCause }		
Detailed Comments :		

(b) Constraint Declaration Table

Figure 9.4: AUTOLINK script language – Using functions in constraint rules

```
TRANSLATE TC_EndInd
  IF $2 == "basic" THEN
    CONSTRAINT NAME "C_" + $0 + "Basic"
               PARS $1 = "Dialog_Id"
  END
  CONSTRAINT NAME "C_" + $0
END
```

Figure 9.5: AUTOLINK script language – A conditional constraint rule

```
TRANSLATE TC_ErrorInd
  CONSTRAINT NAME "C_" + $0 + $3
             PARS $1 = "Invoke_ID", $2 = "Dialog_ID"
             MATCH $4 = "*"
END
```

(a) Constraint rule

ASN.1 ASP Constraint Declaration		
Constraint Name	:	C_TC_ErrorIndTRUE(Invoke_ID : InvokeIDtype; Dialog_ID : DialogIDtype)
ASP Type	:	TC_ErrorInd
Derivation Path	:	
Comments	:	
Constraint Value		
{ invokeIDtype1 Invoke_ID, dialogIDtype2 Dialog_ID, boolean3 TRUE, errorType4 * }		
Detailed Comments :		

(b) Constraint Declaration Table

Figure 9.6: AUTOLINK script language – Constraints with wildcards

The rule in figure 9.5 translates signals of the type *TC_EndInd* into parameterized constraints called *C_TC_EndInd_Basic* if the second signal parameter equals *basic.* Otherwise, the unconditioned section is evaluated, i.e., a constraint with name *C_TC_EndInd* is created (without constraint parameters).

If the exact value of a constraint parameter is irrelevant or unpredictable during test execution, the corresponding signal parameter value should be replaced by a wildcard during test generation. This can be achieved with a `MATCH` statement as shown in figure 9.6(a). The fourth parameter of signals of type *TC_ErrorInd* denotes an error type. Since different errors might occur during test execution, it is replaced by "*" in the TTCN-2 test suite (figure 9.6(b)). The application of TTCN matching mechanisms is only valid for receive events. Hence, `MATCH` statements are only allowed for signals that become receive events in TTCN.

Constraint rules also allow to introduce test suite parameters and constants. By defining a test suite parameter/constant, a concrete signal parameter value in a constraint table is replaced by a symbolic name. The name is listed either in the TTCN *Test Suite Constant Declarations* table (where the value of the substituted signal parameter is assigned to it) or in the *Test Suite Parameter Declarations* table (where an optional reference to a PICS/PIXIT proforma entry is made).

In figure 9.7(a), the rule of figure 9.2 has been extended in such a way that the value of the first signal parameter is replaced by a global test suite parameter called *PIX_DialogId.* If more than one *TC_ContinueReq* signal is translated during test generation and the signals have different values for the first parameter, conflicts are resolved by appending a sequence number to the test suite parameter name.

```
TRANSLATE TC_ContinueReq
  CONSTRAINT NAME "C_" + $0
             PARS $1 = "Dialog_ID"
  TESTSUITE  PARS $1 = "PIX_DialogId"
END
```

(a) Constraint rule

Test Case Dynamic Behaviour					
...					
3		SCF ! TC_ContinueReq	C_TC_ContinueReq(PIX_DialogId1)		
...					
Detailed Comments :					

(b) Test Case Dynamic Behaviour Table

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
PIX_DialogId1	INTEGER		
Detailed Comments :			

(c) Test Suite Parameter Declarations Table

Figure 9.7: AUTOLINK script language – Test suite parameters

The example demonstrates that constraint parameterization and test suite parameterization can be applied to the same signal parameter. As a result, a constraint declaration table for signal *TC_ContinueReq(51,oSCF)* looks essentially the same as with the rule in figure 9.2. However, the value 51 is replaced by *PIX_DialogId1* in all constraint references and an entry is added to the *Test Suite Parameter Declarations* table (see figure 9.7(b) and (c)). Test suite constants can be introduced in the same way as test suite parameters except that the keyword `CONSTS` has to be used instead of `PARS` (for a complete description of the AUTOLINK script language syntax see figure 9.12).

9.1.3 Test Suite Structure Rules

In TTCN-2, test cases and test steps can be combined in *test groups*. A test group is a logical unit that denotes some common property among its members. For example, a test group may comprise test cases that all aim at testing a particular physical or logical unit of some IUT. Alternatively, a test group may describe the kind of testing (stress testing, performance testing, etc.). Test steps can be subsumed, e.g., under two test groups for preambles and postambles.

A test group itself can be part of another one which results in a hierarchy of test groups. In that case, the test groups on the different levels should classify the test cases and test steps according to orthogonal criteria.

A test specifier should have a clear notion of the test suite structure before he starts specifying test purposes, i.e., he should first determine what should be tested and how the tests can be categorized.

When using AUTOLINK, test purposes are described by MSCs. The AUTOLINK script language allows to define rules that place test cases and test steps[3] in different test groups automatically, depending on the names of their corresponding MSCs. These *test suite structure rules* do not only save a lot of manual editing if the test suite has to be regenerated. Even if the test suite is created just once, writing a small script can be beneficial since a single rule can apply to several test cases. In the best case, one test suite structure rule is sufficient to describe the structure of a complete test suite.

In the INAP CS-2 case study, test cases have been classified according to four different criteria: On the top level, tests are distinguished by the system interface that is investigated. On the next level, tests are grouped with regard to the protocol components (common sets) involved. A further subdivision is made based on procedures, i.e., collections of elementary INAP operations that are tested together. Finally, tests are provided for four different test categories.

The structuring of the INAP CS-2 test suite is reflected in the names of the test purposes as shown in figure 9.8. Each name is composed of four acronyms that place the test purpose uniquely in the test group hierarchy. In addition, a sequence number is used to distinguish test purposes belonging to the same test group. According to the scheme in figure 9.8, *IN2_A_BASIC_AT_CA_01* is a valid name. It denotes a test that checks the basic capability of the *ActivityTest* procedure. During test generation, MSC test purposes are mapped to test cases with identical names.

In a simple AUTOLINK script, a separate test suite structure rule can be defined for each path of test groups. The rule in figure 9.9 makes AUTOLINK place the test cases *IN2_A_BASIC_AC_BV_03* and *IN2_A_BASIC_AC_BV_04* in a hierarchy of test groups, with *ValidBehavior* being the innermost one. The names of the test groups in the path specification are separated by a slash ('/') in accordance with the notation of test group references in TTCN-2.

Rules like the one described above can be applied to MSCs with arbitrary names. At best, one test suite structure rule has to be written for each test group. On the other hand, the MSC names in the given example adhere to a regular pattern where some fields of the pattern directly relate to test groups. Thus, instead of a list of concrete names, such a pattern can be specified in the header of a test suite structure rule. As a side effect, several rules can be merged.

Patterns in the AUTOLINK script language are specified identically to *glob patterns* used for file name substitution in command shells (see Gilly, 1994, page 43). The following characters have a special meaning when used in the header of test suite structure rules:[4]

- '*' matches any string of zero or more characters.

[3] In the following, no distinction is made between test cases and test steps as they are handled equally.
[4] Patterns can also be specified in the header of constraint rules and in conditions.

Test purpose name pattern:

IN2_*i*_*sss*_*pp*_*cc*_*nn*

i	=	interface		
		A	≡	SSF-SCF interface
		B	≡	SSF-SRF interface
		C	≡	SCF-SCF interface
sss	=	common set		
		BASIC	≡	Basic set for CS-1 complemented for CS-2
		CPH	≡	Call Party Handling from CS-2
		CTM	≡	Cordless Terminal Mobility from CS-2
pp	=	procedure name		
		AC	≡	ApplyCharging
		AT	≡	ActivityTest
		CA	≡	Cancel
		...	≡	...
cc	=	test category		
		CA	≡	Capability tests
		BV	≡	Valid behavior tests
		BI	≡	Invalid behavior tests
		BO	≡	Inopportune behavior tests
nn	=	sequence number		
		01 – 99		

Figure 9.8: Test purpose naming scheme for INAP CS-2

```
PLACE IN2_A_BASIC_AC_BV_03 | IN2_A_BASIC_AC_BV_04
  IN "BasicSetCS1" / "ApplyCharging" / "ValidBehavior"
END
```

Figure 9.9: AUTOLINK script language – A simple test suite structure rule

```
PLACE "IN2_" + "?" + "_" + "*" + "_" + "*" + "_" + "*" + "_*"
  IN sss(@4) / pp(@6) / cc(@8)
END

FUNCTION sss
    $1 == "BASIC" : "BasicSetCS1"
  | $1 == "CPH"   : "CallPartyHandling"
  | $1 == "CTM"   : "CordlessTerminalMobility"
End

FUNCTION pp
    $1 == "AC"    : "ApplyCharging"
  | $1 == "AT"    : "ActivityTest"
  | TRUE          : "UnknownService"
End

FUNCTION cc
    $1 == "CA"    : "Capability"
  | $1 == "BV"    : "ValidBehavior"
  | $1 == "BI"    : "InvalidBehavior"
  | $1 == "BO"    : "InopportuneBehavior"
End
```

Figure 9.10: AUTOLINK script language - A generalized test suite structure rule

- '`?`' matches any single character.
- '`[...]`' matches any single character in the enclosed list.

  Within `[...]`, character ranges are specified by a pair of characters separated by '`-`'. For example, `[a-z]` matches any lowercase letter. If the first character is '`!`', the expression matches any character *not* enclosed in the brackets.

When a test suite structure rule is evaluated, it is checked first whether one of the terms following the keyword `PLACE` matches the name of the given MSC. The first statement in figure 9.10 has one term consisting of 9 elements that are concatenated with the '`+`' operator. The splitting of patterns makes it possible to refer to a particular element (called *atom*). The '`@`*n*' operator is used to denote the concrete value of the *n*th atom when an MSC name matches the pattern.

Additionally, functions can be defined that map parameters to arbitrary texts. For example, function *sss* in figure 9.10 returns the text "*BasicSetCS1*" if the first parameter passed to *sss* is equal to "*BASIC*".

The application of the test suite structure rule in figure 9.10 and its helper functions is illustrated in figure 9.11. First, AUTOLINK checks whether the pattern of the test suite structure rule matches the given MSC name (*IN2_A_BASIC_AC_BV_03*). At the same time, each atom of the pattern is associated with a substring of the MSC name. After successful matching, the forth atom ("*") is bound to *BASIC*. Therefore, the call of function *sss* with parameter *@4* returns *BasicSetCS1*.

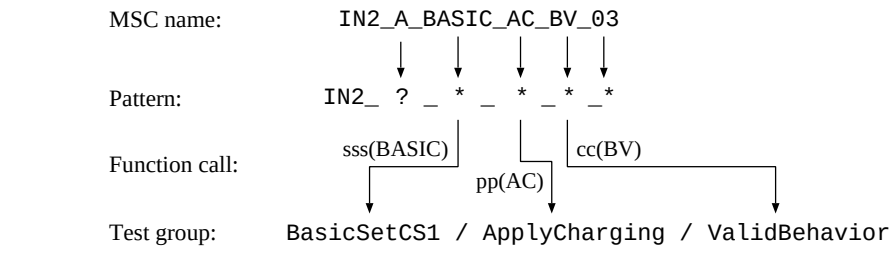


Figure 9.11: Application of the generalized rule

9.2 Automatic Structuring of Constraint Descriptions

The AUTOLINK script language presented in the previous section allows to customize a test suite in a semi-manual way. Among others, it provides the user with a facility to define rules for the parameterization of constraints. While this approach enhances the readability of the generated test suite at low effort, it still requires the intervention of the test specifier.

Preferably, test suites should be optimized automatically. But for that purpose, one needs to define a function $opt : Testsuite \mapsto \mathbb{R}^+$. The optimization of some test suite ts means to find a semantically equivalent test suite ts' so that $opt(ts'') \geq opt(ts')$ is true for all semantically equivalent test suites ts''. In this context, semantical equivalence among test suites is defined by trace equivalence. Obviously, there might be more than one optimal test suite.

But what does optimization mean with regard to abstract tests suites, i.e., what is a reasonable function *opt*? ETSI Technical Report 141 (1994) lists 7 factors that contribute to the overall quality of an ATS and 10 quality criteria that influence these factors (see figure 9.13). In the given context, only the quality factors *usability*, *maintainability*, *testability*, *flexibility*, and *reusability* are relevant, because the correctness should not be influenced.

The given quality criteria are largely subjective and thus cannot be formalized. The size of a test suite might be considered as a weak approximation for an optimization function, i.e., a minimal test suite (in terms of symbols) is supposed to be optimal. On the other hand, a compressed document is not preferable per se. Parameterized entities may be of advantage sometimes but too much parameterization confuses the reader. When computing the "size" of a test suite, each language element may be given a weight. E.g., a formal parameter of a constraint contributes to the size with a higher factor than a constraint name. However, the choice of appropriate weights tends to be arbitrary.

A more promising approach than trying to define a global optimization function is to set up a list of design requirements. Then, a test suite is "optimized" by a sequence of operations that transform it into another one that meets these requirements. Design

```
<configuration>            ::=  {      <constraint rule>
                                |   <structure rule>
                                |   <function> }*
<constraint rule>          ::=  TRANSLATE [ SIGNAL ] <alt term list>
                                { <constraint body if> }* [ <constraint body> ]
                                END
<constraint body if>       ::=  IF <conditions> THEN <constraint body> END
<constraint body>          ::=  [ CONSTRAINT <translation constraint> ]
                                [ TESTSUITE <translation testsuite> ]
<translation constraint>   ::=  [ NAME <term> ]
                                [ PARS <parameter list> ]
                                [ MATCH <parameter list> ]
<translation testsuite>    ::=  [ CONSTS <parameter list> ]
                                [ PARS <ext parameter list> ]
<parameter list>           ::=  <parameter> { , <parameter> }*
<parameter>                ::=  $ <number> [ = <term> ]
<ext parameter list>       ::=  <ext parameter> { , <ext parameter> }*
<ext parameter>            ::=  $ <number> [ = <term> ] [ / <term> ]
<structure rule>           ::=  PLACE <alt term list>
                                { <structure body if> }* [ <structure body> ]
                                END
<structure body if>        ::=  IF <conditions> THEN <structure body> END
<structure body>           ::=  IN <term> { / <term> }*
<function>                 ::=  FUNCTION <identifier> <mappings> END
<mappings>                 ::=  <mapping> { | <mapping> }*
<mapping>                  ::=  <conditions> : <term>
<term>                     ::=  <atom> { + <atom> }*
<atom>                     ::=     $ <number>
                                |  @ <number>
                                |  "text"
                                |  <identifier>
                                |  <function call>
<function call>            ::=  <identifier> ( <seq term list> )
<seq term list>            ::=  <term> { , <term> }*
<alt term list>            ::=  <term> { | <term> }*
<conditions>               ::=  <condition> { AND <condition> }*
<condition>                ::=     <term> { == | != } <term>
                                |  TRUE
```

Figure 9.12: Syntax of the AUTOLINK script language in EBNF

		Quality factors						
		Usability	Correctness	Maintainability	Testability	Flexibility	Portability	Reusability
Quality criteria	Traceability		x					
	Completeness		x					
	Consistency		x	x				
	Simplicity			x	x			
	Generality					x		x
	Instrumentation				x			
	Self-descriptiveness			x	x	x	x	x
	Operability	x						
	Training	x						
	System independence						x	x

Figure 9.13: Quality factors and criteria (ETSI TR 141, 1994)

decisions concern, among others, naming conventions, the test suite structure, the use of timers, and the definition of default trees. With regard to constraints, the test specifier must decide whether or not to use ASN.1, base/modified constraints, static/dynamic chaining, and parameterized/constant constraints. Among others, ETSI TR 141 (1994) proposes the following rules:

1. "Define different base constraints for the send- and receive direction of a PDU (when applicable)." (p. 36)
2. "Use modified constraints preferably when only a small number of fields or parameter values are altered with respect to a given base." (p. 36)
3. "When modified constraints are used, keep the length of the derivation path small. [...] a length greater than 2 is normally difficult to overview and maintain." (p. 36)
4. "Make a careful evaluation of which embedded PDUs are needed in ASPs/PDUs [...] to find an appropriate balance between the use of static and/or dynamic chaining in a particular ATS." (p. 37)
5. "Make a careful overall evaluation of which field/parameter values are needed in ASPs and PDUs to find an appropriate balance between the aim of a comparably small number of constraint declarations and readable and understandable constraint references." (p. 39)
6. "Keep the number of formal parameters small. [...] a number bigger than 5 normally cannot be handled very well." (p. 39)

7. "Constraints used in test steps should be parameterized for simple adaptation of the test steps to specific test cases (p. 49)."

ETSI TR 141 also notes that some ETSI projects exclude the use of modified constraints in their ATS design document. Unfortunately, some of these rules are specified vaguely or are even opposing (2 vs. 7).

The optimization of a test suite cannot be a linear process in the sense that the optimization function is decreased with each transformation. Instead, it is necessary to perform some steps that seem to drift away from the optimal solution but will allow significant improvements in the following. Due to the large size of a test suite document, the extend to which the effect of different transformations is examined, must be strictly limited, leading to the well-known *hill-climbing* problem and suboptimal solutions.

In the following, a simple yet efficient approach to the automatic structuring of constraint descriptions is presented that takes into account all kinds of constraint structuring, i.e., constraint chaining (statically and dynamically), constraint parameterization, and constraint derivation (cf. section 3.1.3). For the reasons explained above, an *optimization* of a test suite in the strict meaning is not possible. Instead, the presented algorithm may only lead to suboptimal yet sufficiently good solutions.

The principle idea is to split complex constraints into a set of chained constraints first and to combine these constraints afterwards. In detail, the algorithm comprises four major steps:

1. Factorization of subconstraints
2. Merging of identical constraints
3. Combination of similar constraints
   a) Constraint parameterization
   b) Constraint derivation
4. Defactorization of constraints

In each of these steps, some general design decisions and protocol-specific constraints are taken into account.

9.2.1 Constraint Factorization

The constraint structuring starts by splitting up complex constraints that consist of nested, structured values and transforming them into a set of chained constraints.

Provided that in the initial state, constraints are neither parameterized nor derived from another constraint, this transformation can be formalized as follows: As long as there is a constraint c in the test suite with a structured constraint value $\{d_1, \ldots, d_i, \ldots, d_n\}$ where d_i itself is a structured value, add a constraint c' with a new, unique name and d_i as its constraint value to the test suite. Furthermore, replace d_i in the value description of c by a reference to c' ($\Rightarrow \{d_1, \ldots, c'(), \ldots, d_n\}$).

After the termination of this process, complex constraints are replaced by an increased number of smaller and simpler constraints which can be handled more flexibly in the succeeding steps. However, in some cases the test specifier might want to control the constraint factorization by one or more options:

1. A list of (structured) data types for which constraints are not to be factorized

   If a nested, structured constraint value forms an indivisible logical unit, all subvalues should be kept in the same constraint declaration.

2. A threshold for the size of constraint values which are to be factorized

   If the size of a constraint in its textual representation is too small, readability decreases. Experiments have shown that the textual description of a constraint value in ASN.1 format should comprise at least 8 words (equals to an ASN.1 sequence with 4 fields) for constraint factorization to become reasonable.

3. A list of (structured) data types for which constraints are to be factorized (disregarding the size threshold)

4. A maximum depth for constraint chaining

   If constraints are chained, a test engineer has to browse through several tables in order to view the complete data structure. While horizontal splitting, i.e., having two or more references on the same level in a constraint, is not problematic, deeply nested constraints have a negative impact on readability. ETSI's TTCN-2 style guide (1994) recommends a maximum nesting depth of 3.

Conditions 2 and 4 can be realized in different way. For example, for a constraint with nesting depth n and a single data field on each level, there are already $\binom{n}{m}$ possibilities to factorize its substructures such that a maximum chaining of m constraints is obtained. To apply the restrictions above in a technically simple and unambiguous way, the factorization process should start at the leaves, i.e., the deepest substructures – if possible – are factorized first.

A practical problem of factorization is that for each constraint that is defined in a test suite, a corresponding data type must be defined as well. Just like nested constraints, nested data types may be specified in a single declaration. In that case, the types for the data elements are declared inline and no name is given to them. Thus constraint factorization may also necessitate data type factorization.

9.2.2 Constraint Merging

In the second step, constraints with identical values are merged. As long as there are two constraints c_1 and c_2 with identical constraint value, c_2 is removed and all references to c_2 are replaced by references to c_1.

A simple algorithm to merge constraints is to take one constraint after the other and compare it with all other constraints (this can be done in $O(n^2)$). However, the order in

which attempts to merge constraints are made is important. Otherwise, some possible mergers might not be detected.

In the following, the notation $c_1 \Rightarrow c_2$ is used to describe that the value description of c_1 includes a reference to c_2. Two constraint descriptions c_1 and c_2 are *syntactically equivalent* ($c_1 = c_2$) if their formal parameters, base constraints, and value descriptions are identical. Two constraint descriptions c_1 and c_2 are *semantically equivalent* ($c_1 \equiv c_2$) if c_1 and c_2 are syntactically equivalent after relabeling of their formal parameters and recursive substitution of all constraint references by the values of the corresponding constraints.

Given the definitions above, consider four constraints $c_1, \ldots, c_4$ with $c_1 \Rightarrow c_3$, $c_2 \Rightarrow c_4$, $c_1 = c_2$, and $c_3 \equiv c_4$. If c_1 is compared with all other constraints first, no possible merger is found. However, after combining c_3 and c_4, the constraint references in c_1 and c_2 are updated which results in $c_1 \equiv c_2$. As a consequence, constraints must be processed in such a way, that those belonging to "deep" data structures are taken first for comparison. (I.e., for $c_1 \Rightarrow ... \Rightarrow c_n$, the constraints have to be investigated in reverse order $c_n, ..., c_1$)

Constraint factorization and merging correspond to the problem of finding common subexpressions in compiler construction. With these steps, it is already possible to, e.g., unify identical PDUs used in different ASPs, even if these ASPs are of different types and hence cannot be merged completely.

9.2.3 Constraint Parameterization

After merging identical constraints, constraints of the same type can be further combined by parameterization. In contrast to the former steps, merging by parameterization does not necessarily minimize the constraint description, as the introduction of formal and actual parameters produces some overhead. This overhead does have to be negative in all cases. If the right fields are chosen as parameters, the reference to a constraint becomes more expressive.

In principle, any set of constraints of the same type can be merged at the cost of parameterization. Given a set C of constraints,

$$\begin{aligned} Part(C) \quad := \{ \quad & \{C_1, \ldots, C_n\} \,|\, \forall i \in \{1 \ldots n\} : C_i \subseteq \mathbb{P}(C), \uplus_{i \in \{1..n\}} C_i = C, \\ & \forall i, j \in \{1..n\}, i \neq j : C_i \cap C_j = \emptyset\} \end{aligned}$$

denotes the set of all possible partitions of C. The number $|Part(C)|$ of all possible partitions is given by the Bell numbers (Graham et al., 1994):

$$\begin{aligned} B(1) \quad &= \quad 1 \\ B(n+1) \quad &= \quad \sum_{k=1}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) B(k) \end{aligned}$$

In an asymptotic approximation, $B(n) \sim n! \frac{e^{e^r - 1}}{r^{n+1}\sqrt{2\pi e^r}}$ with $re^r = n$. For $n = 10$, the number of possible partitions is already $115,975$. Therefore, it is impossible to investigate all possibilities of parameterizing and combining constraints. For the same reason,

branch-and-bound algorithms do not work satisfyingly as the number of constraints for a single type in a test suite can exceed 100.

In order to overcome the problem of complexity, a simpler approach must be chosen that prevents the need to evaluate different possible solutions. A practically applicable approach chosen for the AUTOLINK project is to iteratively try to merge two constraints. After each iteration, it is checked whether the intermediate result, i.e., the parameterized constraints and its references, meet certain design criteria. If the unification indeed improves the constraint representation, the modification is taken over irreversibly. Otherwise, the original constraints are kept and another pair of constraints is investigated.

The run-time complexity of this algorithm is $O(n^2)$ with n being the number of constraints. If two constraints can be merged with probability p, the average number of comparisons for n constraints is equal to

$$\sum_{i=1}^{n}(1-p)^{i-1}(n-i)(1-p)^{i-1} = \sum_{i=1}^{n}(1-p)^{2i-2}(n-i)$$

For $p = 0$, the formula above is equal to $\sum_{i=1}^{n} n - i = n^2 - \frac{n\cdot(n-1)}{2} = \frac{n\cdot(n-1)}{2}$; for $p = 1$, the number of comparisons is $n - 1$.

There are many imaginable criteria (and even combinations of them) that can be used to decide whether parameterization of two constraints is favorable. The chosen approach is to accept all parameterization by default (since one constraint table can be dropped) and to define a list of criteria when parameterization should be prohibited. For example, the test specifier may control the parameterization process by the following options:

- A list of data types for which constraints will never be parameterized
- Prevention of dynamic chaining

  Dynamic constraint chaining, i.e., constraint references inside actual constraint parameters, may result in complex structures. If it is disabled, some constraints might not be mergeable.
- Maximum number of parameters

  An upper limit of 3 parameters has proven to be reasonable.
- Maximum ratio of the parameterized constraint and its references to the two original constraints and their references

  The ratio is measured in words in the ASN.1 definitions and the constraint references. A value < 1 means that the test suite is required to shrink, a value > 1 means that the test suite size is allowed to increase. The default value is 1. To reflect the benefit of reducing the number of constraint tables, a user-defined constant *tabsize* is taken into account when computing the ratio. Its default value is 10.

cref(t_1): a()
cref(t_1): b()
con(t_1): a() := { 1, c() }
con(t_1): b() := { 1, d() }
con(t_2): c() := { 2, 3 }
con(t_2): d() := { 2, 4 }

(a) Original chained constraints and constraint references

cref(t_1): a(**c(3)**)
cref(t_1): a(**c(4)**)
con(t_1): a(**fp:t_2**) := { 1, **fp** }
con(t_2): c(**fp:int**) := { 2, **fp** }

(b) First combine constraints of type t_1, then constraints of type t_2

cref(t_1): a(**3**)
cref(t_1): a(**4**)
con(t_1): a(**fp:int**) := { 1, **c(fp)** }
con(t_2): c(**fp:int**) := { 2, **fp** }

(c) First combine constraints of type t_2, then constraints of type t_1

Figure 9.14: The impact of type ordering on parameterization

- Minimum ratio of the size of the parameterized constraint to the average size of its references

  This option prevents that a large part of the data description becomes actual parameters. The suggested ratio is 4.

- A list of types whose values are not allowed as parameters

  The test specifier may specify data types for which constraint values in both constraints must be identical, i.e., no parameterization is allowed for them. If the value of a data field is crucial for the meaning of a constraint, the corresponding data type should be put on this exclusion list.

A problem with iterative binary parameterization is that the algorithm may produce suboptimal solutions. For example, consider five constraints with $c_1() := \{2, 8, 4, 2, 1\}$, $c_2() := \{3, 8, 4, 2, 2\}$, $c_3() := \{2, 7, 4, 5, 1\}$, $c_4() := \{2, 5, 4, 3, 1\}$, and $c_5() := \{2, 3, 4, 9, 1\}$. If c_1 is merged with c_2, the newly created constraint c_{1+2} is parameterized over the first and last data field. Successive attempts to merge c_{1+2} with c_3 to c_5 will fail because two more parameters would have to be introduced which exceeds the maximum number of parameters. If, on the other hand, c_1 were merged with c_3 first, c_1, c_3, c_4, and c_5 could all be combined at the cost of only two parameters. Methods based on examining different solutions such as branch-and-bound avoid such dead ends but are not applicable in this application context.

In analogy to constraint merging, the order in which constraints of different types are analyzed has an impact on the result. In figure 9.14(a), two constraint references and four constraints are given. Constraints a and b are of type t_1. They both refer to different constraints of type t_2 (c and d respectively). If the constraints of type t_1 are combined first, the embedded constraint references become actual parameters in the constraint references to a and b. If c and d (type t_2) are united in the following, actual parameters are introduced in the constraint references to c inside the constraint references to a and b (see figure 9.14(b)).

If, on the other hand, the constraints of type t_2 are parameterized first, the references in a and b are updated, i.e., both constraints refer to the same new constraint c. As a consequence, if constraints a and b are merged, the references inside of them do not have to become actual parameters as a whole but only the parameters inside the constraint references. Subjectively, the second way of parameterization is preferable. Thus, in analogy to constraint merging (section 9.2.2), constraints that are embedded in other constraints should be considered first. If $c_1 \Rightarrow c_2$, all constraints of the same type as c_2 should be considered for parameterization before all constraints with the same type as c_1 are investigated.[5]

9.2.4 Constraint Derivation

Besides parameterization, constraint derivation is a mechanism to reduce the effort to describe similar constraints. A constraint is called *base constraint* if there is a *modified constraint* that refers to the base constraint. Within the description of the modified constraint only those data fields have to be listed that differ from the corresponding data fields in the base constraint. Accordingly, constraint derivation defines a binary relation between two constraints. Typically, a base constraint describes the "regular" or "ideal" case, while a derived constraint describes a special case. In a test suite, there may be several base constraints of the same data type.

In principle, each pair of constraints of the same type can be checked for whether derivation leads to the desired saving. If the data descriptions of both constraints are identical by a certain percentage, say 75% of all words, constraint derivation might be beneficial. But while constraint parameterization would simply merge the constraints, constraint derivation imposes a semantical relation between them.

Thus, the question is which — if any — of the two constraints represents the "more typical" case. There are two heuristics that differ in whether the base constraints are elements of the set of original constraints and whether the computation is based on statistical analyses.

[5] Though TTCN-2 allows recursive data type definitions, mutual inclusion among different data types is quite unusual. Therefore, it is possible to construct a directed acyclic graph where a node represents a data type and an arc from t_1 to t_2 means that type t_2 is used inside a structured type t_1. During the parameterization process, a type t must be chosen for which no outgoing arc exists. After the constraints of this type have been evaluated, the node for t as well as all arcs to the node are removed from the graph and a new type is selected.

- Base constraint selection based on the number of references

  If there are many more references to one of the two constraints, i.e., the ratio of the references is above a certain threshold, it is very likely that this constraint describes the "normal case".

- Base constraints with default values

  A new base constraint is introduced, whose data fields are set to the values of the two given constraints if they are identical, or set to default values if they differ (e.g., 0 for integers and floating point numbers, *false* for booleans, *empty sequence* for sequences, etc.). The original constraint descriptions are modified in such a way that they refer to the new base constraint. If one of the two constraints is identical to the new base constraint, it is removed from the test suite. This case occurs if the data values of the constraint are either identical to those of the second constraint or represent default values.

Constraint modification can be combined with constraint parameterization. For the prototypical implementation and the case studies in section 9.2.6, constraint derivation has not been considered. The question, whether the two criteria given above can be justified empirically, is subject to further studies.

9.2.5 Constraint Defactorization

In order to be able to combine nested data values, factorization is applied initially. This step leads to a vast number of constraints. After merging and parameterization, constraints might exist that are used only once. These constraints bloat the test suite unnecessarily and do not contribute to enhanced readability.

Therefore, all constraints for which only a single reference exists are resolved in a final step. As a contrary operation to factorization ($\rightarrow$ *de*factorization), all references to these constraints are replaced by the value descriptions of the constraints and the constraints are removed from the test suite. (Of course, only those references can be resolved in TTCN-2 that appear inside the constraint declaration part.)

Surprisingly, a constraint which is used only once may nevertheless be parameterized. For example, constraint c in figure 9.14(c) has a formal parameter although it is used exclusively by constraint a.

Defactorization can be controlled by the following settings:

- A (negative) list of types for which constraints will never be defactorized
- A (positive) list of types for which constraints will always be defactorized

  By convention, the negative list overrules the positive list.

- A maximum size of constraints which are to be defactorized

  The size is measured in words in the ASN.1 definition. By default, the size is set to infinite so that even large constraints are defactorized.

Number of constraints	Orig	Fact	Merge	Param	Defact
VB5.2	71	249	156	53	51
INAP CS-2	324	473	436	200	152

Figure 9.15: Number of constraints in the VB5.2 and INAP CS-2 test suites

Size of test suite	Orig	Fact	Merge	Param	Defact
VB5.2	3,081	3,615	2,416	1,472	1,460
INAP CS-2	18,670	19,117	18,772	17,610	17,454

Figure 9.16: Size of the VB5.2 and INAP CS-2 test suites

After defactorization, all constraints should be renamed. In particular, if sequence numbers are used to distinguish constraints of the same type, this resolves gaps in the enumeration.

9.2.6 Case Studies

The applicability and benefits of automatic constraint optimization have been demonstrated by a prototypical implementation that performs the four processing steps factorization, merging, parameterization and defactorization. The prototype is based on a PROLOG system (Zhou, 2000) and makes use of the list manipulation language described in the next section. It takes a test suite in list form (provided by a modified version of AUTOLINK) as input and returns a TTCN MP file as output.

The prototype has been applied to test suites derived from the SDL specifications of VB5.2 and Core INAP CS-2. The VB5.2 case study is based on the same test suite as the one that has been generated by AUTOLINK for ETSI project STF 151. It comprises 41 test cases and 4 test steps. For INAP CS-2, a test suite consisting of 136 test cases and 69 test steps has been created.

Figures 9.15 and 9.16 present two statistics on how the original test suites have changed after each processing step. In figure 9.15, the number of constraints is presented. Starting with 71 and 324 constraints in the original test suites, the number increases significantly during factorization. In particular, many constraints are created for the VB5.2 test suite, as the protocol makes use of deeply nested ASN.1 data types. However, in the following steps, the number of constraints decreases again. In case of INAP CS-2, the final test suite comprises less than half as many constraints as in the original one. This reduction was made possible mainly by merging 191 constraints of the same type into 39 parameterized constraints.

The size of each test suite is given in figure 9.16. It is measured in words in the TTCN MP file where all keywords and special characters (e.g., brackets, colons) have been removed.

The declarations part is not considered, either, as it did not change during the constraint transformations. Surprisingly, there is no correlation between the number of constraints and the size of a test suite. In the VB5.2 case study, the size of the test suite that is obtained by factorization and merging of identical constraints is reduced by about 21 percent. At the same time the number of constraints doubles. After defactorization, the test suite is reduced by more than 50 percent, even though the number of constraints has decreased only moderately. This phenomenon is explained by the fact that the PDUs of the VB5.2 protocol have several fields in common. Regarding the INAP CS-2 test suite, there is only a slight improvement in terms of the size even though a lot of constraints could be merged.

When it comes to the assessment of the resulting test suites, no objective judgment can be made. However, discussions with another member of STF 151 gives evidence that the automatic constraint structuring is a big step in the right direction. An example of how VB5.2 constraints have been transformed is given in figures 9.17 and 9.18.

9.3 A List Pattern Matching and Manipulation Language

The automatic structuring of test suites presupposes that the various transformation steps are formalized in an algorithmic way. In the extreme, these transformations are fully specified in a common programming language and tightly coupled with the internal representation of test suites. However, this leads to a complicated code, in particular, if the transformation is to be integrated in an application that does not provide a clear API. Moreover, it is difficult to maintain and sensitive to changes of the overall data structure.

A more flexible approach is to regard a test suite as a structured document. A structured document can be represented as a tree where each intermediate node bares structuring information and the leave nodes denote the actual content of the document. Provided that all relevant semantic information is given explicitly in the tree, a test suite manipulation is equal to a tree transformation.[6]

Transformation of Structured Documents. Kuikka and Penttonen (1995) distinguish between different types of transformations: *Structure-preserving transformations* result only in differences in the leaves of the tree. *Local transformations* operate on a part tree. E.g., (nonterminal) children of a nonterminal node are re-ordered. Finally, *global transformations* involve several part trees with long distance dependencies. Obviously, test suite restructuring falls in the latter category since, for example, changing the name of a constraint in the constraint part of a test suite requires the update of all of its references in the dynamic behavior part.

[6] Theoretically, a graph is more appropriate for describing the relations within a structured document. However, graph transformations are much more difficult to describe. On the other hand, relations that do not fit to the hierarchical structure of trees can be encoded (syntactically) by additional, redundant nodes.

ASN.1 PDU Constraint Declaration	
Constraint Name	: cMODIFY_COMP_REJ_1
PDU Type	: ModifyCompRej
Derivation Path	:
Encoding Rule Name	:
Encoding Variation	:
Comments	:

Constraint Value

```
{
  commonMsgInfo {
    protDiscr '49'H, transId {
      transIdLength '03'H, transIdFlag '1'B, transIdVal 20
    },
    msgType '56'H, msgCompatInd '80'H, msgLength '0000'H
  },
  rejCauseIE {
    commonIEInfo {
      iEType '07'H, iECompatInd '80'H, iELength '0000'H
    },
    rejCauseOctet 1
  }
}
```

Detailed Comments :

ASN.1 PDU Constraint Declaration	
Constraint Name	: cMODIFY_COMP_REJ_2
PDU Type	: ModifyCompRej
Derivation Path	:
Encoding Rule Name	:
Encoding Variation	:
Comments	:

Constraint Value

```
{
  commonMsgInfo {
    protDiscr '49'H, transId {
      transIdLength '03'H, transIdFlag '1'B, transIdVal 21
    },
    msgType '56'H, msgCompatInd '80'H, msgLength '0000'H
  },
  rejCauseIE {
    commonIEInfo {
      iEType '07'H, iECompatInd '80'H, iELength '0000'H
    },
    rejCauseOctet 7
  }
}
```

Detailed Comments :

Figure 9.17: Original VB5.2 constraints

ASN.1 Type Constraint Declaration	
Constraint Name	: c_CommonMsgInfo_1(p_transIdFlag_1 : BIT_STRING; p_transIdVal_2 : INTEGER; p_msgType_3 : OCTET_STRING)
ASN1 Type	: CommonMsgInfo
Derivation Path	:
Encoding Variation	:
Comments	:
Constraint Value	
{ protDiscr '49'H, transId { transIdLength '03'H, transIdFlag p_transIdFlag_1, transIdVal p_transIdVal_2 }, msgType p_msgType_3, msgCompatInd '80'H, msgLength '0000'H }	
Detailed Comments :	

ASN.1 Type Constraint Declaration	
Constraint Name	: c_RejCauseIE(p_rejCauseOctet_1 : INTEGER)
ASN1 Type	: RejCauseIE
Derivation Path	:
Encoding Variation	:
Comments	:
Constraint Value	
{ commonIEInfo { iEType '07'H, iECompatInd '80'H, iELength '0000'H }, rejCauseOctet p_rejCauseOctet_1 }	
Detailed Comments :	

ASN.1 PDU Constraint Declaration	
Constraint Name	: cMODIFY_COMP_REJ(p_transIdVal_1 : INTEGER; p_rejCauseOctet_2 : INTEGER)
PDU Type	: ModifyCompRej
Derivation Path	:
Encoding Rule Name	:
Encoding Variation	:
Comments	:
Constraint Value	
{ commonMsgInfo c_CommonMsgInfo_1('1'B, p_transIdVal_1, '56'H), rejCauseIE c_RejCauseIE(p_rejCauseOctet_2) }	
Detailed Comments :	

Figure 9.18: Chained and parameterized constraints for VB5.2

Unfortunately, no self-contained formalism exists for describing all kinds of global transformations. Instead, a heterogeneous approach has to be chosen where some aspects are specified elegantly in a special, yet restricted, notation and the overall transformation algorithm is written in a universal programming/script language that allows to perform arbitrary operations. (Compare this to the processing of text documents by a PERL script that uses regular expressions as a compact notation for search patterns.)

The transformation of structured documents involves two steps:

- Searching for a particular pattern (with a given context) in the document.
- Replacing those parts of the document that match the pattern.

In traditional text editing tools, these two operations are specified separately. For example, the well-known UNIX tools SED and PERL use the notation `s/`X`/`Y`/` to denote that some string X (described by a regular expression) is to be replaced by Y. However, if the pattern shall only be replaced in a certain context, this results in redundant and error-prone duplication. E.g., in order to replace `a` by `b` only if surrounded by C_1 and C_2, SED/PERL require to write `s/(`C_1`)a(`C_2`)/\1b\2/` where `\`n is equal to the sequence of characters that matched with the nth parenthesized subexpression.

Moreover, there is no possibility to make use of the programming language from within the embedded pattern language. Thus, if a search pattern contains alternatives or iterations and the replacement depends on what text has actually been scanned, the expense (in terms of code size) to specify the transformation increases unnecessarily.

Evidently, document processing based on tree transformations is more powerful than substitutions of character strings. On the other hand, it is more difficult to locate one or all parts of a document (i.e., subtrees) that match a given pattern. Various operators are needed to specify which parts of the tree are to be inspected.

In the following, a general-purpose pattern matching and manipulation language is presented. It simplified considerably the development of the prototype described in the previous section. The *List Pattern matching and Manipulation Language* (henceforth, *LPML* for short) can be applied to nested lists or trees[7] where the nodes are decorated with untyped constants. It provides many built-in operators for both describing tree patterns and basic tree transformations where the latter are specified directly as part of a larger pattern.

9.3.1 General Language Concepts

The LPML is based on the unification concept and data model of PROLOG (PROgramming in LOGic, see Clocksin and Mellish, 1994), a language that is wide-spread in the research area of artificial intelligence.

In PROLOG, all data are represented by *terms*. A term is either an atomic constant, a variable, or a structure. Constants are names (starting with lowercase letters) or numbers. Variables are represented by names with initial capital letter. Finally, structures

[7] Flat and nested lists are special variants of trees; each tree can be transformed into a nested list and vice versa.

take the form $f(t_1, t_2, \ldots, t_n)$ where f is a functor and $t_i, i \in \{1 \ldots n\}$ are terms again. Empty lists are denoted by the special constant '[]'. Non-empty lists are represented by binary structures with functor **"."**. The term $.(H, R)$ denotes a list with H as the first element and R as the rest of the list. For convenience, $.(H, R)$ can be written as $[H|R]$ and a list with n elements, i.e., $.(e_1, .(e_2, .(\ldots .(e_n, [\,]) \ldots)))$, can be described by the syntactic shorthand $[e_1, e_2, \ldots, e_n]$.

In logic programming, the concept of variable assignments as known from imperative languages has been replaced by *unification* of arbitrary terms. A variable v can be bound to/unified with some term t_1 but the binding of v to a different term t_2 afterwards will fail unless t_1 can be unified with t_2. As a consequence, it is not allowed to modify existing ground (variable-free) terms. Similarly, an element cannot be deleted from an existing list. Instead, a new list must be created that is identical to the former one except for the missing element.[8]

According to these concepts, an interpreter for the LPML checks whether a pattern matches a given list. If it does, the interpreter returns a *new* list with all transformations incorporated, and constant `fail` otherwise. The signature of an LPML interpreter is match(*Pattern*, *InputList*, *OutputList*), where *Pattern* must be a term conforming to the LPML, *InputList* must be a ground term only consisting of constants and list structures, and *OutputList* must be a unbound variable which is bound to a term with constants, list structures, and variables after successful pattern matching.

The LPML provides various constructs that can be divided into five categories:

- Basic terms for forming simple patterns, including wildcards and operators for the description of alternatives and iterations
- Variables and operators for retrieving information and recognizing recurring patterns
- Operators for manipulating lists
- Operators for searching and extracting information whose exact position within a nested list is unknown
- An extension operator for complex transformations that are outside the scope of the LPML

In the following, each group is described in detail. A complete overview of the LPML syntax is given in figure 9.19.

9.3.2 Basic Patterns

A ground pattern in the LPML is either a constant (atomic name or number) or a list consisting of ground patterns. Due to the recursive definition it is possible to specify

[8] In general, such operations do not cause a relevant memory overhead as different data structures "share" common subexpressions. This structure sharing is safe due to the fact that ground terms cannot be modified.

```
<definition>   ::=  { <variable> = <pattern> , }* <pattern>
<pattern>      ::=  [ <variable> - ]
                    {     <token>
                        | <list>
                        | not ( <pattern> )
                        | or ( <pattern> , <pattern> )
                        | scope ( <pattern> )
                        | collect ( <term> , <term list> , <pattern> )
                        | eval ( <external goal> )
                        | test ( <pattern> )
                        | repeat ( <variable> , <pattern> )
                        | iterate ( <term> , <term list> , <pattern> )
                        | iteratetest ( <term> , <term list> , <pattern> )
                        | replace ( <pattern> , <term> )
                        | replacelist ( <bindvar> , <pattern list> , <term> )
                        | delete ( <pattern> )
                        | find ( <term> , <bindvar> , <pattern> )
                        | findtop ( <term> , <bindvar> , <pattern> )
                        | findalltop ( <term list> , <bindvar list> , <pattern> )
                        | findallrec ( <term list> , <bindvar list> , <pattern> ) }
<list>         ::=  '[' [ <element> { , <element> }* ] ']'
<element>      ::=     <pattern>
                    | partlist ( { <list> | <variable> - <list> | <variable> } )
                    | insert ( <term> )
                    | alt ( <element>, <element> )
                    | opt ( <element> )
                    | star ( <element> )
                    | plus ( <element> )
<token>        ::=     any
                    | <constant>
```

Figure 9.19: Syntax of the LPML in EBNF

nested lists of arbitrary depth. A ground pattern only matches itself. For example, the term `[a, [b, c]]` represents a pattern that matches a list with two elements where the first element is the constant `a` and the second element is a list itself consisting of the constants `b` and `c`.

The special atom `any` can be used as a wildcard that matches any atom or list. Negation is expressed by the `not` operator.

The LPML supports all operators known from regular expressions and string pattern matching. These operators can be used to describe alternatives (`or`/`alt`), optional elements (`opt`), and the (positive) closure of elements (`plus`/`star`). Except for `or`, they are only allowed to occur *within* a list, i.e., they have list *elements* as parameters.

Two different operators are provided for expressing alternatives: `or(`*pattern*`,` *pattern*`)` can only be used for describing two alternative lists, whereas the `alt(`*element*`,` *element*`)` operator may have partial lists, i.e., subsequences of list elements, as its parameters. Partial lists are denoted by `partlist(`*list*`)`. They are only allowed to be specified inside a list. Examples: The pattern `or([a], [b])` matches with both `[a]` and `[b]`. The pattern `[a, alt(partlist([b, c]), [b, c]), d]` matches with `[a, b, c, d]` and `[a, [b, c], d]`.

The pattern `[`e_1`, ..., `e_{i-1}`, opt(`e_i`), `e_{i+1}`, ..., `e_n`]` with e_i being some list element matches both `[`e_1`, ..., `e_{i-1}`, `e_{i+1}`, ..., `e_n`]` and `[`e_1`, ..., `e_i`, ..., `e_n`]`. Operator `star(`e`)` matches what zero or more consecutive occurrences of element e would match. For instance, the pattern `[star([a, opt(b)])]` matches any list whose elements are again lists that have `a` as their first element, followed by an optional atom `b` (e.g., `[[a], [a, b]]`). `plus(`e`)` is a syntactic shorthand; the pattern `[..., plus(`e`), ...]` is equivalent to `[..., `e`, star(`e`), ...]`.

9.3.3 Variables and Variable Operators

In order to keep track of what has been recognized, any (sub-)pattern can be prefixed by a variable. When a pattern is recognized successfully, the variable is bound to the list that matches the pattern. Typically, a variable is used in conjunction with a pattern that includes the token `any`. For example, if pattern `[a, V-any]` is applied to list `[a, [b]]`, variable `V` is equal to `[b]`.

Variables are bound when their corresponding pattern is applied for the first time. If the same pattern is evaluated another time, the input must be the same as before. Therefore, variables can be used to ensure that substructures recur identically. Example: Pattern `[star(V-any)]` matches any list whose elements are uniform (e.g., `[a, a, a]` or `[[b], [b]]`).

Variables may also be used inside the `partlist` operator where they are bound to a sequence of list elements. This extension allows for back references for multiple elements. For instance, `[partlist(V-[a, b]), partlist(V)]` matches with `[a, b, a, b]`.

By default, all variables are defined within a global scope. Unfortunately, this is inadequate for checking whether an input list is in $L_1 = \{[[x_1, x_1], [x_2, x_2], \ldots, [x_n, x_n]] \mid x_i \in$

Constant, $n \in \mathbb{N}\}$. For instance, the pattern `[star([V-any,V-any])]` only matches words in $L_2 = \{[[x,x],\ldots,[x,x]] \mid x \in \mathit{Constant}\}$. In order to express that a new copy of `V` shall be created each time within the `star` operation, the `scope` operator can be used. For L_1, a conforming pattern is `[star(scope([V-any,V-any]))]`. Precisely, `scope` creates new instances for all variables in its embedded pattern that are unbound at the time of evaluation; if a variable is bound outside the scope before (such as in `[V-any,star(scope(V-any))]`), the `scope` operator will have no effect.

If a pattern shall apply to all elements of a given list and information on each element shall be collected, the `iterate` operator can be used. Its general form is `iterate(`*term*`,` *termlist*`,` *pattern*`)` where *term* contains variables used inside *pattern* and *termlist* is an originally unbound variable. For each list element (that must match with *pattern*), a new instantiation of *term* is added to *termlist*. Example: `iterate(T,TL,[a,T-any])` matches with `[[a,b],[a,c]]` and TL is bound to `[b,c]`.

If a pattern does not have to match with all list elements, operator `iteratetest(`*term*`,` *termlist*`,` *pattern*`)` can be used alternatively. It matches with any list and *termlist* contains information only on those list elements that match with the given pattern. `iteratetest` can be used with manipulation operators to keep track of when a transformation was possible.

In order to be able to perform static well-formedness checks, the LPML forbids the usage of unbound variables inside patterns except where indicated in the grammar (see figure 9.19). As a consequence, it is impossible to describe recursive patterns in the form `X-[a,X]`. However, the pattern `repeat(X,[a,X])` may be used instead. Moreover, variables may be bound to patterns prior to the main pattern to describe recursion that involve more than one variable and to make the LPML as expressive as regular tree grammars. For instance, the pattern `X=or(c,[a,Y]),Y=[b,X],X` matches with `c`, with `[a,[b,c]]`, with `[a,[b,[a,[b,c]]]]`, and so on.

9.3.4 Manipulation Operators

The LPML provides three basic operators for manipulating lists. An `insert` operator can be used as a list element. It matches the empty input and adds a new term to the output list. Thus, if pattern `[any,insert([in,between]),any]` is applied to list `[a,b]`, an LPML interpreter will return `[a,[in,between],b]` as output list.

The `delete` operator can be used to remove an element from a list. `delete(`p`)` matches with any input with which p matches as well. If pattern `[a,delete(any),b]` is applied to `[a,[in,between],b]`, the output list is `[a,b]`. For technical reasons, `delete` replaces the matching input list by a reserved constant `null_pattern`. However, this constant is only visible to the user if the operator is used on top-level where it does not make sense. If `delete` is used *inside* a pattern, `null_pattern` is removed from the output list.

Operator `replace` substitutes a list by an arbitrary term. For instance, `[a,replace([b],b),c]` matches with `[a,[b],c]` and produces output list `[a,b,c]`.

`delete` and `replace` can be used in combination with variables in order to log what has actually been deleted or replaced. For example, given the pattern `[a,replace(V-any,d),c]` and the input list `[a,b,c]`, an LPML interpreter will not only return `[a,d,c]` as output list but also bind variable `V` to `b`.

For convenience, `replacelist` can be used as a syntactic shorthand if different kinds of patterns shall be replaced. `replacelist(`b`,[`$p_1, \ldots p_n$`],`t`)` is semantically equivalent to `replace(`b`-or(`p_1`,or(...,or(`p_{n-1}`,`p_n`)...),`t`)`.[9]

If the input list shall be modified if it matches a pattern but remain unchanged otherwise, the `test` operator can be used. For instance, `[a,test(replace(b,d)),c]` results in output list `[a,d,c]` if applied to `[a,b,c]` but it also matches with, e.g., `[a,e,c]`. In fact, `test(`*pattern*`)` is equivalent to `or(`*pattern*`,any)`. It should be noted that using `test` inside `iterate` does not make sense in general, because `iterate` would try to collect information even in the failure case. Instead, `iteratetest` should be used.

9.3.5 Extension Operator

The LPML is not a closed language in the sense that any desired transformation can be expressed within the LPML itself. Complex manipulations may depend on the evaluation of user-defined predicates. While XSLT (Clark, 1999) has a predefined set of operations, LPML provides an operator called `eval` which has an external goal as its argument. `eval` matches the empty input but only under the condition that the goal is satisfied. Otherwise pattern matching fails.

9.3.6 Search Operators

One of the strengths of the LPML are its search operators. A search operator is used for finding a given pattern within a larger input list. It allows to manipulate substructures easily in cases where

- the exact position of the substructure within the list is unknown.
- it is complicated and error-prone to specify the position because the format of the input list is subject to regular changes.
- several substructures match the search pattern that are located at different positions in the input list.

The general structure of a search operator is *search-op(term, bindvar, pattern)*. *search-op* matches a list if *pattern* matches the list or any of its sublists. In case of success, the (sub-)list matching the pattern is bound to the variable *bindvar* and replaced by *term* in the output list. In total, there are four operators for finding patterns: `find`, `findtop`, `findallrec`, and `findalltop`. They differ with regard to two criteria:

[9] Obviously, a similar shorthand could be defined for `delete` as well.

		Inspection of sublists if main list matches		
		During Backtracking	At the same time	No inspection
Number of solutions	One	*find*	—	*findtop*
	All	—	*findallrec*	*findalltop*

Figure 9.20: Classification of search operators

- Number of solutions

  A search operator may either apply to one or all occurrences of the search pattern. In the first case, the input list is traversed in preorder, i.e., a (sub-)list is checked for matching before its elements are checked. If the application of the search operator is successful, but pattern matching fails for the complete pattern, back-tracking takes place and another (sub-)list that matches the pattern is searched for. This procedure is comparable to the one of the `star` operator (section 9.3.2).

  If a search operator applies to all occurrences of the search pattern, *term* and *bindvar* are lists where the nth list matching the pattern is bound to the nth element of *bindvar* and replaced by the nth element of *term*. Since the exact number of solutions is not known in advance in most cases, an unbound variable must be specified for *term list* and *bindvar list*.

- Inspection of sublists

  A pattern may match with a list and one or more of its sublists at the same time. The search operators can be classified accordingly by whether they explore the whole list recursively or leave the current list as soon as the search pattern matches with it.

Figure 9.20 shows how the search operators of the LPML fit into the scheme above. To illustrate their differences, the operators are applied to three sample lists in figure 9.21.

The operators defined above always traverse all substructures to find a list or token that matches the search pattern. For large lists, this procedure may be too costly. On the other hand, depending on the application context it might be known in which sublists a search pattern will *never* match. The efficiency of the search operators could be improved by specifying an additional *blocking pattern*. If some list l matches the blocking pattern, the LPML interpreter will not search for the pattern inside of l. Another possibility to restrict the scope of the search operators is to limit the search depth.

9.4 Discussion

In this section, two ways have been presented for improving the readability of test suites gained by automatic test generation: Transformations by means of user-defined rules and automatic constraint structuring with no or only little guidance by the test specifier. The first approach was realized in the AUTOLINK tool and applied in various

Application 1: Find occurrences of a given pattern (distinction between `find`/`findtop` and `findalltop`/`findallrec`)

```
Input list:    Inlist = [u, [a, b], v, [a, c], w]
Pattern:       Pat = [a, any]

Invocations:   match(find(Var, BindVar, Pat), Inlist, Outlist)
               match(findtop(Var, BindVar, Pat), Inlist, Outlist)
Results:       BindVar = [a, b]
               Outlist = [u, Var, v, [a, c], w]

Invocations:   match(findalltop(Var, BindVar, Pat), Inlist, Outlist)
               match(findallrec(Var, BindVar, Pat), Inlist, Outlist)
Results:       Var = [Var1, Var2]
               BindVar = [[a, b], [a, c]]
               Outlist = [u, Var1, v, Var2, w]
```

Application 2: Find occurrences of a pattern that is also used at another place (distinction between `find` and `findtop`)

```
Input list:    Inlist = [u, [a, [a, [b]]], v, [a, [b]], w]
Pattern:       Pat = [a, any]

Invocation:    match([u, find(Var, BindVar, Pat), v, BindVar-any, w],
                     Inlist, Outlist)
Result:        BindVar = [a, [b]]
               Outlist = [u, [a, Var], v, [a, [b]], w]

Invocation:    match([u, findtop(Var, BindVar, Pat), v, BindVar-any, w],
                     Inlist, Outlist)
Result:        matching fails
```

Application 3: Find all occurrences of a given pattern recursively (distinction between `findalltop` and `findallreq`)

```
Input list:    Inlist = [u, [a, [a, [b]]], v]
Pattern:       Pat = [a, any]

Invocation:    match(findalltop(Var, BindVar, Pat), Inlist, Outlist)
Result:        Var = [Var1]
               BindVar = [[a, [a, [b]]]]
               Outlist = [u, Var1, v]

Invocation:    match(findallrec(Var, BindVar, Pat), Inlist, Outlist)
Result:        Var = [Var1, Var2]
               BindVar = [[a, [b]], [a, Var1]]
               Outlist = [u, Var2, v]
```

Figure 9.21: Application of the search operators

case studies at ETSI; for the latter, a prototype has been implemented and applied to test suites of Core INAP CS-2 and VB5.2. Furthermore, a language has been described and implemented that allows for complex syntactic transformations of lists.

Although the AUTOLINK script language and the automatic constraint structuring are independent developments, their combined use could further enhance the readability of test suites. For instance, constraint rules could be applied to the new constraints created by factorization.

Autolink Script Language. Despite its simplicity, the script language has proven to be sufficiently powerful. Nevertheless, it can easily be extended by built-in functions that perform any kind of text manipulation such as transforming all alphabetic characters of a text into lower-case characters.

One restriction of the current implementation is that only top-level signal parameters can be specified, i. e., it is not possible to address nested parameters. One possible solution is to support parameter references in the form `$x.y.z` with $x, y, z \in \mathbb{N}$. A more convenient notation is `$n.subpar.subsubpar` with $n \in \mathbb{N}$ and *subpar*, *subsubpar* being symbolic field names.[10]

The presented approach to the translation of SDL signals to TTCN-2 constraints is based on signal types, i.e., all transformations (parameterization, replacement of values by matching mechanisms, etc.) are specified for particular parameters inside particular signal types. Alternatively, a data type-oriented approach might be chosen. For example, a rule might state that a signal parameter value of type T always becomes a test suite parameter, regardless of where T is actually used. Of course, such signal-independent rules can only be defined if all values of a specific type are handled equally. The question, which approach is better in meeting the way of thinking of a test specifier and results in more compact scripts, is subject to further studies.

Automatic Structuring of Constraint Descriptions. The presented prototype for automatic constraint structuring is closely coupled with AUTOLINK. But enhancing the quality of a test suite is not only a concern for automatic test generation. It is also a matter for manual test suite development. Hence, a module for the structuring of constraints should operate on top of the API of a general-purpose TTCN tool.

At the Institute for Telematics, a parser has been developed that provides access to the abstract syntax tree of a given TTCN-3 test suite (see Schmitt et al., 2001). An adapted version of the presented prototype (along with an adapted version of the LPML) that operates on such an AST would be able to give hints to the user on how to restructure the constraints (called *templates* in TTCN-3) or transform a test suite automatically according to some style guide.[11]

[10] The numeric notation has to be kept on the top level, since SDL does not define field names for signal parameters!

[11] Due to the reference semantics of Java/C# and the object-oriented features of C++, most concepts of the LPML can be ported easily to these programming languages.

The presented prototype is used in a batch mode. If the test specifier is not satisfied with the output, he can modify the settings and re-run the constraint optimizer until finally the desired structure is generated. However, interactive, semantics-preserving test suite manipulations from inside a TTCN editor where the test specifier controls each single processing step, would also be of great benefit.

In contrast to its predecessor, TTCN-3 allows to define constraints within a test case itself, i.e., it is not necessary to refer to a constraint that is defined separately. While this allows for compact test case descriptions, the increased flexibility also involves potential inconsistencies. Therefore, it becomes even more important to be able to restructure an existing test suite automatically.

10 Advanced Test Generation by Symbolic Execution

When it comes to simulation, validation, and test generation, one of the most challenging tasks is to model the environment of the system specification. SDL tools like the Tau Validator or Autolink trigger the system by signals whose parameters are composed of predefined values. In addition, they allow the test specifier to define a fixed set of input signals. However, it is often impossible to foresee all relevant signals. As a consequence, automatic test generation may lead to tests with only a low system coverage.

One possible solution to this problem is the symbolic execution of specifications. Instead of specific signals, all inputs are represented by symbolic values. Concrete values are computed in retrospect at the end of an execution path. Symbolic execution is classified as a static, structural test in literature (Coward and Ince, 1995, page 12). Nevertheless, symbolic execution proceeds in the same way as regular program execution except that values are computed symbolically.

When traversing along a particular path, a path condition is computed. Whenever a choice point is reached, e.g., in terms of an `if` statement, the corresponding condition is added to the path condition by conjunction. Thereafter, it is checked whether the extended path condition is still satisfiable, i.e., whether there is a variable assignment such that the path condition holds. If a path condition cannot be fulfilled, an invalid state is reached and the path must be pruned. By solving the path condition, conclusions can be made about previous input values. That means at the end of a path, it is known which inputs have to be used in order to execute this particular path.

Symbolic execution is not a new techniques. Actually, the idea of executing a program with symbolic values can be traced back to Boyer et al. (1975) and Hantler and King (1976). Systems for various languages were built since then and a few people have suggested using symbolic execution also for test generation based on formal description techniques (see, e.g., Touag and Rouger, 1999). Nevertheless, symbolic execution has gained hardly any relevance in practice. One explanation why symbolic execution failed in the past is the effort required to check the feasibility of path conditions. However, in the recent decade, the computational power has increased significantly. At the same time, efficient algorithms have been developed for constraint satisfaction and optimization problems.

In this chapter, the ValiBOSE (VALIdation Based On Symbolic Execution) tool is presented that has been developed by the author. Although still being a prototype, it already allows to get an impression on how test generation can benefit from symbolic execution. In section 10.1, a few shortcomings of traditional test generation tools

are described and the use of symbolic execution is motivated. Section 10.2 provides an overview of techniques for checking the feasibility of path conditions, with focus on algorithms that operate on variables with finite domains. The ValiBOSE tool is presented in section 10.3. Finally, open issues are discussed in section 10.4.

10.1 Motivation

The simulation of modern telecommunication systems and protocols suffers from two problems. These are:

- the missing or insufficient modeling of the system environment and
- the state space explosion problem.

Both problems are tightly related to each other. When simulating an SDL specification, there are no general rules for what signals should be used as input. By default, tools like the Tau Validator/Autolink set up some predefined values for each data type, e.g., -55, 0, and 55 for integers. Signals with multiple parameters are composed of combinations of these default values. The choice of values is more or less arbitrary. A positive value, zero, and a negative value may be suitable in some cases. But in other cases, a sequence like 1,2,3,... might be preferable.

Alternatively, the test specifier can define a fixed set of test signals. But due to the complexity of the interaction of a system or protocol with its environment, even an expert cannot foresee all relevant inputs. Moreover, it may even be theoretically impossible to set up a *finite* set of signals that allow to cover all possible paths.

On the other hand, different signals may effect the control flow of the system in exactly the same way. As a consequence, the state space exploration becomes inefficient. Symbolic execution partitions test input data in equivalence classes with regard to their impact on the control flow. Similarly, the states of the state space are partitioned.

Test generation based on traditional simulation has a few shortcomings. For illustration purposes, a simple MSC test purpose for the Inres protocol is shown in figure 10.1. Although the MSC describes a valid test purpose, it is not optimal:

1. No information on data flow and dependencies

   The MSC does not express that the parameter of signal *IDATreq* (i.e.,*0*) is passed through the SUT without modifications and is output as the third parameter of signal *MDATind*. Ideally, the causal connection between the input and output values should be indicated explicitly.

   Moreover, it not perceptible that the sequence number sent with signal *MDATreq(AK,?)* must always be identical to the sequence number received by the preceding signal *MDATind(DT,?,?)*. Although they are identical in *this* example (*zero*), no generalization can be made.

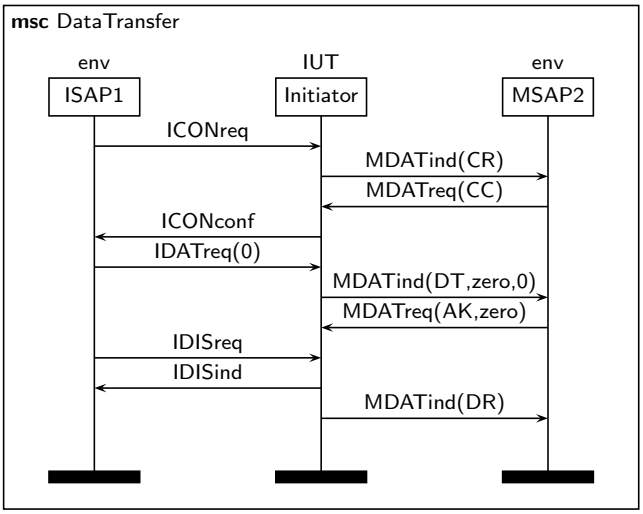


Figure 10.1: A simple, harmful MSC test purpose

2. The necessity of test case variables cannot be detected

   As mentioned above, the tester must respond with the same sequence number that has been received before. As a consequence, a test generation tool should create a test case in which the content of signal *MDATind* is stored in a test case variable. Unfortunately, it is impossible to deduce the need for a test case variable from the MSC test purpose in figure 10.1.

3. Erroneous test verdicts due to "overspecification"

   The successful execution of a test case that is derived from the given MSC test purpose depends on whether the SUT sends *zero* as sequence number or not. If it sends *one*, the test case results in a fail verdict erroneously.

4. Repetitive test execution affects the test verdict

   Since the sequence number changes with every data transmission, a second test case run will always fail if the first one was successful.

There are more cases where a symbolic execution might be beneficial. For example, traditional simulators cannot handle SDL externals (that map to PIXIT parameters). Instead, concrete (dummy) values must be used. Since these values may influence the control flow, a generated test case may only be valid for this particular PIXIT value. Moreover, from a methodological point of view, it is inappropriate to use a concrete value first and to replace it by a parameter in a test case description afterwards. A test generation tool based on symbolic execution is able to cope with parameterized specifications and might even be able to create test selection expressions automatically.

10.2 Checking the Feasibility of Path Conditions

The key problem of symbolic execution is finding a solution for path conditions. Theoretically, a path condition can become arbitrarily complex. The decidability and complexity of a feasibility test is determined by the variable domains and the types of conditions.

Definition 8 (Constraint System) *A* constraint system *is a triple* $CS = \langle \Sigma, CT, \mathcal{C} \rangle$ *where*

- Σ is a signature that contains at least the null-ary symbols *true* and *false*.
- CT is a nonempty, consistent constraint theory over Σ.
- $\mathcal{C}$ is a set of valid constraints over Σ such that at least $true \in C$ and $false \in C$.

Typically, constraint systems are based on terms, boolean algebra, finite domains (FD), linear equation systems over real or rational numbers, or non-linear equation systems. In case of boolean algebra, a constraint system is defined in the following way: $\Sigma = \{true, false\}$, CT describes the boolean algebra itself, and $\mathcal{C}$ is the set of all constraints that can be constructed by variables and the operators $=$, $\neg$, $\wedge$, and $\vee$.

There are many different ways to solve constraints/path conditions. If the path condition is expressed by or transformed into first degree predicate logic, a resolution calculus can be applied (see Schöning (1989) for details). The interactive validation tool SMILE (Eertink, 1994) uses a related approach, called *narrowing*, for the evaluation of LOTOS data specifications. While this approach is generally applicable, resolutions have a poor performance in general.

In contrast, linear equation systems can be solved efficiently by the Gaussian elimination algorithm (Bronstein et al., 1999, p. 885). For linear inequality systems, the Simplex procedure can be used which is also applied to optimization problems (Bronstein et al., 1999, p. 846ff). Nonlinear equation systems can be solved by, e.g., the Gröbner bases algorithm, interval arithmetic, and an extension of Newton's approximation procedure for finding null points (Frühwirth and Abdennadher, 1997, p. 119ff). Unfortunately, these approaches do not necessarily describe a solution uniquely. Instead, it can only be approximated, i.e., for any variable, an interval is determined in which one or more solutions are included. In this context, an important aspect is the termination criteria that describes the minimal size of such an interval.

Constraint Satisfaction Problems over Finite Domains

Efficient algorithms have been elaborated for constraint systems based on variables with finite domains (FD). Since the size of a constraint and thus also the number of variables involved is finite, FD problems are decidable. In the worst case, the solvability of a constraint satisfaction problem (CSP) is decided in exponential time by enumeration of all possible variable assignments. In practice, however, a solution (or the non-existence of a solution) can be determined more efficiently.

SDL-2000 does not specify the value ranges of integers and the precision of floating point numbers. Nevertheless, for simulation purposes it is legitimate to assume that their domains are finite. A concrete implementation that conforms to the specification will also rely on some encoding rules that restrict the domains.

A constraint satisfaction problem can be represented by a (hyper-)graph. A hypergraph is a tuple $H = \langle V, E \rangle$, where V is a set of nodes and $E \subseteq V^*$ is a set of hyperedges that connect an arbitrary number of nodes. Each node represents a distinct variable. It is labeled with the potential values that the variable can take without violating a constraint. The edges denote the constraints and link all nodes/variables that are involved in the constraint. In principle, every constraint satisfaction problem can be represented by unary and binary constraints only, so that it is sufficient to consider graphs (Tsang, 1996, page 10).

Solving a CSP means finding an assignment for each variable such that all constraints are satisfied (arc/node consistency). A CSP is unsatisfiable if the domain of some variable becomes empty.

An efficient approach to solve CSPs is based on two principles: domain reduction and constraint propagation. *Domain reduction* means that all values are eliminated from the variable domain which conflict with some constraint. *Constraint propagation* is the process of computing the consequences of a domain reduction on other variables. If a value is removed from the domain of some variable V, all variables that have a constraint in common with V, are investigated as well. If their domains can also be reduced, constraint propagation takes place recursively.

Domain reduction and constraint propagation are performed in polynomial time. However, they cannot decide alone whether a CSP is satisfiable or not. Therefore, they are typically combined with a search strategy that recursively chooses a non-instantiated variable (with $|dom(V)| > 1$) and restricts its domain to one of the possible values. After each instantiation, constraint propagation is triggered which restricts the search space significantly. If a variable instantiation leads to the violation of a constraint, another value is chosen from the domain during back-tracking.

The efficiency of the search algorithm depends on a number of factors:

- The order of variable instantiations

  The next variable to be instantiated should be determined dynamically at each choice point. Preferably, the most difficult variable should be chosen first (*fail-first principle*). Experience has shown that most often this is the one with the smallest domain. (ILOG, 1999b, p. 119)

- The choice of values for instantiation

  If only one solution is needed, the order in which values are selected from the variable domain is important. Values might be chosen at the lower or upper domain boundary. In case of integers, a value near zero might be preferable.

- The connectivity of the graph

  If a graph is unconnected, i.e., there are variables that do not influence each other, then each subgraph can be examined independently.

Many improvements have been proposed in literature. For instance, if a constraint is violated, the set of variables that are involved in the conflict can be determined. If it turns out that the instantiation of the previous variable(s) is irrelevant, then this information can be used to break the regular back-tracking scheme. Real numbers which have a very large domain can be handled by interval partitioning instead of instantiation. An excellent overview of CSP techniques is given by Barták (1996), a comprehensive description can be found in Tsang (1996).

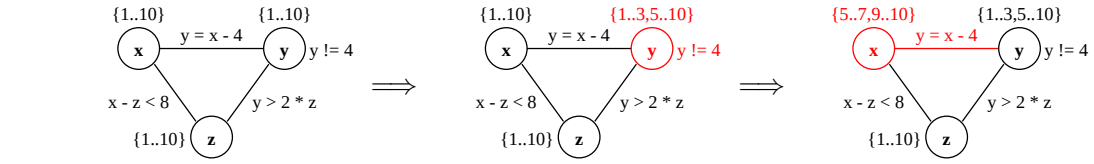
Example. The efficiency of constraint propagation and domain reduction and its superiority to handwritten code is demonstrated by a simple example. Given three variables x, y, and z with $dom(x) = \{1 \dots 10\}$, $dom(y) = \{1 \dots 10\}$, and $dom(z) = \{1 \dots 10\}$. Find all assignments to these variables for which the following (in-)equations hold:

$$\begin{aligned} y &= x - 4 \\ x - z &< 8 \\ y &\neq 4 \\ y &> 2 \cdot z \end{aligned} \tag{10.1}$$

A graph $G = (N, E)$ is constructed with nodes $N = \{x, y, x\}$ and edges $E = \{(x, y), (x, z), (y, z), (y, y)\}$. Initially, node consistency is achieved for node y by excluding the value 4 from the variable domain (10.2(2)). Next, arc consistency is established for all non-unary constraints: From $y = x - 4 \Leftrightarrow x = y + 4$ and $dom(y) = \{1 \dots 3, 5 \dots 10\}$, the conclusion can be drawn that the domain of x is restricted to at most $dom(x) = \{5 \dots 7, 9 \dots 10\}$ (10.2(3)). Moreover, the maximum value of y can be no higher than 6 since $max(dom(x)) = 10$ (10.2(4)). Constraint $y > 2 \cdot z$ and $min(dom(z)) = 1$ imply $y > 2$ and thus $dom(y) = \{3, 5 \dots 6\}$ (10.2(5)). In the other direction, $y > 2 \cdot z \Leftrightarrow z < \frac{y}{2}$ and $max(dom(y)) = 6$ limits the domain of z to $\{1, 2\}$ (10.2(6)).

The domain reduction for y in the one but last step makes it necessary to check arc consistency for constraint $y = x - 4$ with regard to the domain of x again. As a result, values 5 and 6 are removed from $dom(x)$ (10.2(7)). Hereafter, constraint $x - z < 8$ is evaluated. Since $max(dom(z)) = 2$, the equation $x < 8 + 2 \Leftrightarrow x < 10$ must hold, i.e., 10 can be excluded from the domain of x as well. (10.2(8)). This again triggers a consistency check for $y = x - 4$ and the domain of y is reduced to $\{3, 5\}$ (10.2(9)). After this last step, consistency is achieved among all arcs in the graphs. However, the fact that each variable still has a non-empty domain does not necessarily imply the existence of a valid assignment.

To find all concrete solutions, a choice point is introduced and all possible values of x are considered separately. After setting $x := 7$ (10.2(10a)), constraint propagation takes place again. Variable y is dependent on x by constraint $y = x - 4$ and thus its domain



(1) Initial state (2) Node: $y \neq 4$ (3) Arc: $y = x - 4$

(4) Arc: $y = x - 4$ (5) Arc: $y > 2 \cdot z$ (6) Arc: $y > 2 \cdot z$

(7) Arc: $y = x - 4$ (8) Arc: $x - z < 8$ (9) Arc: $y = x - 4$

(10a) Choice point: $x := 7$ (10b) Arc: $y = x - 4$ (10c) Arc: $y > 2 \cdot z$

(11a) Choice point: $x := 9$ (11b) Arc: $y = x - 4$ (11c) Arc: $x - z < 8$

Figure 10.2: CSP example

gets restricted to value 3 only (10.2(10b)). This again enforces a consistency check for $y > 2 \cdot z$ and causes the domain of z to be set to $\{1\}$ (10.2(10c)). Since $x - z < 8$ holds for $x = 7$ and $z = 1$, arc consistency is achieved. Moreover, a single value is assigned to each variable. Thus computation stops at this point and $(x = 7, y = 3, z = 1)$ denotes a solution for the given constraint problem.

If x is set to 9 (10.2(11a)), arc consistency for constraint $y = x - 4$ is preserved only by $dom(y) = \{5\}$. This time, constraint $y > 2 \cdot z$ does not affect the domain of z. However, constraint $x - z < 8$ implies $z > x - 8 = 9 - 8 = 1$ and thus $dom(z) = \{2\}$. Once again, the domains of all variables have exactly one value. $(x = 9, y = 5, z = 2)$ denotes an alternative valid assignment that fulfills the equations given above.

10.3 The ValiBOSE Tool

In the ValiBOSE (VALIdation Based On Symbolic Execution) project, the applicability of symbolic execution for validation and test generation has been examined in practice. CSP techniques have been applied for checking the feasibility of path conditions and computing concrete test data. The ValiBOSE tool makes use of the ILOG Solver (ILOG, 1999a), a commercial C++ library for solving constraint satisfaction problems.

A prototypical simulator was developed that takes a sequential program (written in a subset of the action language of SDL-2000 extended by an *input* operation to read variables) and transforms it into an extended finite state machine. The state space of the EFSM can then be explored either in a step-wise manner or fully automatically by backtracking.

An important aspect of symbolic execution is its ability to provide the user with a detailed insight in the functionality of a system, e.g., by showing the dependencies between input and output data. On the other hand, the user may be flooded by useless information. Therefore, special attention was drawn to the presentation of simulation results.

10.3.1 Navigation

The state space of a specification can be explored either in an interactive or automatic manner. For manual navigation, ValiBOSE provides commands to execute single transitions, to move up the current path by undoing the last transition, or to jump back to the start state immediately.

To simplify navigation in case of long sequences of unconditional code, a command is available that executes all transitions up to the next choice point. Branches for which it can be proven by normalization (see below), i.e., without the constraint satisfaction engine, that their condition evaluates to false, are automatically removed from the list of possible transitions. This allows users to even skip `if` statements whose conditions are not dependent on external inputs. Similarly, `for` statements can be executed in a single step.

Automatic state space exploration can be performed by means of depth first searches. ValiBOSE supports iterative deepening where the increment between two iterations can be specified by the user. During the exploration and at its end, the user is provided with statistics about the number of executed transitions, the achieved coverage, the maximum depth, etc.

10.3.2 Coverage Measurements

ValiBOSE supports two control-flow based coverage criteria: branch coverage (C1 coverage) and loop coverage (Cx coverage).

For each transition, the total number of executions is recorded. Thus, branch coverage (C1 coverage) is easily determined by computing the ratio between transitions that have been executed at least once and the total number of transitions. Transitions that have never been executed might be unreachable and indicate a programming error. ValiBOSE helps to identify such potentially dead code.

For loop coverage detection, two processing steps are needed: When a specification is loaded, ValiBOSE performs a control flow analysis and marks all transitions that denote the end of a loop. During simulation, two loop counters are maintained for these transitions: The first one counts the number of times a loop is executed along the current path. It is incremented whenever the final loop transition is executed and decremented during backtracking. The second, global counter is independent from the current path and stores the maximum value that the first counter has ever reached.

10.3.3 Assertions

ValiBOSE allows to specify `assert` statements in a specification. They define invariants of the specification in terms of a boolean expression and can be used for formal proofs. During simulation, ValiBOSE tries find a variable assignment so that the negation of the assertion is fulfilled. If such an assignment exists, a *Violation* bookmark is created (see below).

10.3.4 Bookmarks

In order to quickly browse through the state space of a specification, ValiBOSE allows to set *bookmarks*. If the user visits a state of particular interest, he can define a bookmark that allows him to directly jump to that state from any other state in the reachability graph. For that purpose, a bookmark contains the full path, i.e., all transitions, from the initial state to the target state.

ValiBOSE automatically generates various types of bookmarks during both interactive and automatic state space exploration. These bookmarks are triggered by state transitions and their corresponding actions. *BranchCoverage* bookmarks are created whenever a transition is executed that has never been executed before. *LoopCoverage* bookmarks

are stored if the n-th iteration ($n \leq 5$) of a loop was completed for the very first time. If a condition with some external variable (i.e., a symbolic input value) is infeasible, a *Contradiction* bookmark is set. The failure of an assertion is marked by a *Violation* bookmark. Finally, a *Termination* bookmark indicates the end of program execution.

The number of bookmarks generated during automatic state space exploration can become very large. Therefore, a user can view compressed statistics (number of bookmarks, minimum and maximum path length for each category), print a list of all bookmarks along with their triggering actions, and remove unneeded bookmarks. For convenience, Message Sequence Chart diagrams can be produced for one or all bookmarks of a specific type without actually having to go to their target state.

10.3.5 Normalization

Normalization of variable terms and path conditions is an important means to improve the readability of the user output and to accelerate the internal processing. During symbolic execution, the algebraic expressions which represent the variable values become more and more complex. In particular, this holds for loop variables which are incremented with each iteration and result in terms like `1+1+1+1+1+...+1`. ValiBOSE provides the user with two representations:

- The original expression whose structure reflects the order of computation.
- A normalized expression that is equivalent to the original expression.

For performance improvements, former normalization results are cached so that each term (or even a subterm inside a larger term) has to be normalized only once. Complex terms cause a significant performance loss and do not improve comprehension. Therefore, normalization is triggered as soon as the size of a term surmounts a threshold of 10 symbols (variables, literals, or operators). Then, the original expression is replaced by the normalized one automatically.

10.3.6 Test Data Selection

For each path, ValiBOSE does not only present the path condition. Instead, it is also able to print information on valid variable instantiations. If there is more than one solution, the user can choose whether minimal or maximal values shall be assigned to the variables. As a third alternative, values close to zero may be retrieved. The test data selection criterion can be changed at any time during simulation.

10.3.7 Example

The application of the ValiBOSE tool is demonstrated by a simple specification that models a system for access control. Its code is shown in figure 10.3.

When a user is in front of some imaginary gate, the access control system asks him to enter his user identification (line 12). In the given specification, only two user IDs

```
1:  {
2:     dcl pinUserA, pinUserB, failureNoA, failureNoB INTEGER;
3:     dcl stateNo, userID, key, pin, pinOK, failureNo INTEGER;
4:
5:     pinUserA := 209; failureNoA := 0;
6:     pinUserB := 873; failureNoB := 0;
7:     stateNo := 1;
8:
9:     for ( , , ) { /* Endless loop */
10:       decision ( stateNo ) {
11:            ( 1 ) :  {
12:                        output( 'Please enter your ID.' );
13:                        input( userID );
14:                        decision ( userID ) {
15:                            ( 1 ) :  { pinOK := pinUserA; failureNo := failureNoA; }
16:                            ( 2 ) :  { pinOK := pinUserB; failureNo := failureNoB; }
17:                             else :  failureNo := 2;
18:                        }
19:                        if ( failureNo <= 1 ) {
20:                           output( 'Please enter your PIN.' ); stateNo := 2; pin := 0;
21:                        } else {
22:                           output( 'Invalid user ID.' );
23:                        }
24:                     }
25:           ( 2:4 ) :  {
26:                        input( key );
27:                        decision ( key ) {
28:                            ( -2 ) :  { output( 'Authentication is canceled.' ); stateNo := 1; }
29:                            ( -1 ) :  if ( stateNo > 2 ) {
30:                                         output( 'Last PIN digit is deleted.' );
31:                                         pin := pin / 10; stateNo := stateNo - 1;
32:                                      }
33:                            ( 0:9 ) :  { pin := pin * 10 + key; stateNo := stateNo + 1; }
34:                        }
35:                     }
36:            ( 5 ) :  {
37:                        if ( pin = pinOK ) {
38:                           output( 'PIN is correct. The door opens.' );
39:                        } else {
40:                           failureNo := failureNo + 1;
41:                           if ( failureNo > 1 )
42:                              output( 'PIN is incorrect (2nd time). Your ID is locked.' );
43:                           else
44:                              output( 'PIN is incorrect. Please retry.' );
45:                           decision ( userID ) {
46:                              ( 1 ) : failureNoA := failureNo;
47:                              ( 2 ) : failureNoB := failureNo;
48:                           }
49:                        }
50:                        stateNo := 1;
51:                     }
52:       }
53:    }
54: }
```

Figure 10.3: ValiBOSE example – Access control

(namely *1* and *2*; lines 15 and 16) are accepted; any other ID is rejected immediately. After entering a valid user ID, the user is requested to enter a 3-digit PIN (line 20). Each key stroke is processed separately (lines 25–35). Besides entering the digits 0 to 9, the user is able to abort the authentication process (line 28) and to undo the last input (lines 29–32) by pressing special keys with represented as values -2 and -1.

As soon as the complete PIN has been entered, the access control system validates it against the expected PIN stored internally (209 for user ID 1, 873 for user ID 2). If both are identical, the system displays a confirmation and the user is allowed to pass the gate (line 38). Otherwise, the user is informed about the incorrect PIN. If, in addition, a wrong PIN has been entered for the second time (see variable *failureNo*), the corresponding user ID is locked (line 42) and successive attempts to authenticate will fail (lines 19 and 22).

The log of a typical ValiBOSE session for the access control example is printed on the following pages. Where suitable, the program input and output is annotated to ease comprehension. The tool demonstration covers most ValiBOSE features including bookmarks, coverage analysis, and various ways to display path information and external variables.

The session starts with loading the specification which is converted into an EFSM. Then, several user commands for manual navigation through the state space are executed until finally the state is reached in which the system grants access. After listing different representations of the execution path and storing the current state as bookmark and MSC, the system is reset into the initial state. Thereafter, an automatic state space exploration is performed until a 90% branch coverage is achieved. From the list of generated bookmarks, the one is chosen that drives the system in the state where it denies access due to an incorrect PIN. Again, different ways to display path information and selecting external variables are illustrated.

```
------------------------------------------------------------------
         VALIBOSE - Validation Based On Symbolic Execution
 Copyright (C) 2001 Institute for Telematics, University of Luebeck
                    M. Schmitt (schmitt@itm.mu-luebeck.de)
                    J. H. Sauselin (valibose@sauselin.de)
------------------------------------------------------------------

Type "help" for more information.

[Valibose]$ efsm.load Tests/access-control.sdl ⇐ load the specification and convert it into an extended finite state machine
Parsing file 'Tests/access-control.sdl'...
ILOG Solver 4.400, licensed to "university-luebeck"

1: Instantiation (1:1)⇐ next possible action with reference to the specification document (line/column information)

[Tests/access-control.sdl]$ transition.down 1 ⇐ execute the first transition

1: Assignment (5:3)         pinUserA := 209

[Tests/access-control.sdl]$ transition.branch ⇐ continue execution until the next branch

1: Decision (15:21)         userID = 1
2: Decision (16:21)         userID = 2
3: Decision (15:21)         not ( userID = 1 ) and not ( userID = 2 )

[Tests/access-control.sdl]$ transition.branch 2

1: Decision (28:21)         key = -2
2: Decision (29:21)         key = -1
3: Decision (33:21)         key >= 0 and key <= 9
4: Decision (28:21)         not ( key = -2 ) and not ( key = -1 ) and not ( key >= 0 and key <= 9 )

[Tests/access-control.sdl]$ transition.branch 3

1: Decision (28:21)         key = -2
2: Decision (29:21)         key = -1
3: Decision (33:21)         key >= 0 and key <= 9
4: Decision (28:21)         not ( key = -2 ) and not ( key = -1 ) and not ( key >= 0 and key <= 9 )

[Tests/access-control.sdl]$ transition.branch 3

1: Decision (28:21)         key = -2
2: Decision (29:21)         key = -1
3: Decision (33:21)         key >= 0 and key <= 9
4: Decision (28:21)         not ( key = -2 ) and not ( key = -1 ) and not ( key >= 0 and key <= 9 )

[Tests/access-control.sdl]$ transition.branch 3

1: If statement (37:24)     pin = pinOK
2: If statement (37:24)     not ( pin = pinOK )

[Tests/access-control.sdl]$ transition.down 1
```

```
1: Enter scope (37:38)
[Tests/access-control.sdl]$ transition.down 1
1: Output (38:21)           output( 'PIN is correct. The door opens.' )
[Tests/access-control.sdl]$ transition.down 1
1: Leave scope (37:38)
[Tests/access-control.sdl]$ output.toggle ⇐disable output of possible actions after each command
[Tests/access-control.sdl]$ path.print
No.  Name                  Code                                 =>  Evaluation                                  =>  Normal form
  0  Instantiation (1:1)
  1  Assignment (5:3)      pinUserA := 209
  2  Assignment (5:20)     failureNoA := 0
  3  Assignment (6:3)      pinUserB := 873
  4  Assignment (6:20)     failureNoB := 0
  5  Assignment (7:3)      stateNo := 1
  8  Loop condition (0:0)  true
 10  Decision (11:7)       stateNo = 1                          =>  1 = 1                                       =>  true
 12  Output (12:19)        output( 'Please enter your ID.' )
 13  Input (13:19)         input( userID )                      =>  input( userID?11 )
 19  Decision (16:21)      userID = 2                           =>  userID?11 = 2
 21  Assignment (16:31)    pinOK := pinUserB                    =>  pinOK := 873
 22  Assignment (16:50)    failureNo := failureNoB              =>  failureNo := 0
 26  If statement (19:24)  failureNo <= 1                       =>  0 <= 1                                      =>  true
 28  Output (20:21)        output( 'Please enter your PIN.' )
 29  Assignment (20:57)    stateNo := 2
 30  Assignment (20:71)    pin := 0
  8  Loop condition (0:0)  true
 37  Decision (25:7)       stateNo >= 2 and stateNo <= 4        =>  2 >= 2 and 2 <= 4                           =>  true
 39  Input (26:19)         input( key )                         =>  input( key?12 )
 53  Decision (33:21)      key >= 0 and key <= 9                =>  key?12 >= 0 and key?12 <= 9                 =>  not ( 0 > key?12 ) and
                                                                                                                    not ( key?12 > 9 )
 55  Assignment (33:33)    pin := pin * 10 + key                =>  pin := 0 * 10 + key?12                      =>  pin := key?12
 56  Assignment (33:56)    stateNo := stateNo + 1               =>  stateNo := 2 + 1                            =>  stateNo := 3
  8  Loop condition (0:0)  true
..  ...                   ...                                      ...                                             ...
 37  Decision (25:7)       stateNo >= 2 and stateNo <= 4        =>  2 + 1 + 1 >= 2 and 2 + 1 + 1 <= 4          =>  true
 39  Input (26:19)         input( key )                         =>  input( key?14 )
 53  Decision (33:21)      key >= 0 and key <= 9                =>  key?14 >= 0 and key?14 <= 9                 =>  not ( 0 > key?14 ) and
                                                                                                                    not ( key?14 > 9 )
 55  Assignment (33:33)    pin := pin * 10 + key                =>  pin := ( ( 0 * 10 + key?12 ) * 10 +         =>  pin := 100 * key?12 +
                                                                          key?13 ) * 10 + key?14                        10 * key?13 + key?14
 56  Assignment (33:56)    stateNo := stateNo + 1               =>  stateNo := 2 + 1 + 1 + 1                    =>  stateNo := 5
```

```
 8  Loop condition (0:0)  true
60  Decision (36:7)       stateNo = 5                      =>  2 + 1 + 1 + 1 = 5                          =>  true
62  If statement (37:24)  pin = pinOK                      =>  100 * key?12 + 10 * key?13 + key?14 = 873
64  Output (38:21)        output( 'PIN is correct.
                                 The door opens.' )
```

```
[Tests/access-control.sdl]$ path.print -o -v ⇐print observable events with concrete values for all external variables
No.  Name                 Code                                    =>  Evaluation    =>  Normal form
 0  Instantiation (1:1)
12  Output (12:19)       output( 'Please enter your ID.' )
13  Input (13:19)        input( userID )                         =>  input( [2] )  =>  input( 2 )
28  Output (20:21)       output( 'Please enter your PIN.' )
39  Input (26:19)        input( key )                            =>  input( [8] )  =>  input( 8 )
39  Input (26:19)        input( key )                            =>  input( [7] )  =>  input( 7 )
39  Input (26:19)        input( key )                            =>  input( [3] )  =>  input( 3 )
64  Output (38:21)       output( 'PIN is correct. The door opens.' )
```

```
[Tests/access-control.sdl]$ path.conditions -v ⇐print all path conditions that involve external variables (with concrete values for the latter)
No.  Name                 Code                 =>  Evaluation
19  Decision (16:21)     userID = 2           =>  [2] = 2
53  Decision (33:21)     key >= 0 and key <= 9  =>  [8] >= 0 and [8] <= 9
53  Decision (33:21)     key >= 0 and key <= 9  =>  [7] >= 0 and [7] <= 9
53  Decision (33:21)     key >= 0 and key <= 9  =>  [3] >= 0 and [3] <= 9
62  If statement (37:24)  pin = pinOK          =>  100 * [8] + 10 * [7] + [3] = 873
```

```
[Tests/access-control.sdl]$ coverage.statistics
40 out of 85 branches covered (47.0588%). Executions: 0-5.
1 out of 1 loops covered at least once (100%). Max iterations: 4-4.
```

```
[Tests/access-control.sdl]$ path.msc access.mpr ⇐store current path as Message Sequence Diagram
```

```
[Tests/access-control.sdl]$ bookmark.add Access ⇐remember the path to the current state
```

```
[Tests/access-control.sdl]$ transition.top ⇐return to the start state of the specification
```

```
[Tests/access-control.sdl]$ statespace.explore -c90 ⇐perform automatic state space exploration
State space exploration will stop after achieving 90% branch coverage.
Starting depth first search with iterative deepening...
Transitions:      0, Max depth:   1, Branch coverage: 47.0588%, Time: Sat Jun  8 05:14:40 2002
Transitions:  25219, Max depth:  64, Branch coverage: 76.4706%, Time: Sat Jun  8 05:14:50 2002
Transitions:  49916, Max depth:  70, Branch coverage: 82.3529%, Time: Sat Jun  8 05:15:00 2002

State space exploration statistics
----------------------------------
Transitions:        72009
Maximum depth:      73
```

```
Computation time:   28 seconds

77 out of 85 branches covered (90.5882%). Executions: 0-5788.
1 out of 1 loops covered at least once (100%). Max iterations: 6-6.

[Tests/access-control.sdl]$ coverage.deadcode ⇐ list all transitions that have never been executed
Transition No. 7:
Loop condition (0:0)
false ⇐ implicit loop condition is always true

Transition No. 69:
If statement (41:26)
failureNo > 1

Transition No. 70:
Output (42:23)
output( 'PIN is incorrect (2nd time). Your ID is locked.' )

...

Transition No. 84:
Termination (1:1)

[Tests/access-control.sdl]$ bookmark.statistics
BranchCoverage     :  77 bookmarks. Length: 1-73.
Contradiction      :   4 bookmarks. Length: 72-72. ⇐ paths leading to an unsatisfiable condition
LoopCoverage_1x    :   1 bookmark.  Length: 26.
...                    . ...        ...     ..
LoopCoverage_5x    :   1 bookmark.  Length: 58. ⇐ a path with 5 loop iterations
UserBookmark       :   1 bookmark.  Length: 69.

Total number of bookmarks: 87

[Tests/access-control.sdl]$ bookmark.print
BranchCoverage
   1:  Depth =  1 - Instantiation (1:1)
   2:  Depth =  2 - Assignment (5:3)         pinUserA := 209
  ..   ...     ..   ...                      ...
  72:  Depth = 71 - Output (44:23)           output( 'PIN is incorrect. Please retry.' )
  ..   ...     ..   ...                      ...
  77:  Depth = 73 - Assignment (47:31)       failureNoB := failureNo
Contradiction
  78:  Depth = 72 - Decision (46:23)         not ( userID = 1 ) and not ( userID = 2 )
  79:  Depth = 72 - Decision (47:23)         userID = 2
  80:  Depth = 72 - Decision (46:23)         not ( userID = 1 ) and not ( userID = 2 )
  81:  Depth = 72 - Decision (46:23)         userID = 1
LoopCoverage_1x
  82:  Depth = 26 - Leave scope (9:15)
...
```

```
LoopCoverage_5x
  86:  Depth = 58 - Leave scope (9:15)
UserBookmark
  87:  Depth = 69 - Access
[Tests/access-control.sdl]$ bookmark.goto 72
[Tests/access-control.sdl]$ path.print -o ⇐print observable events of the current path
No.  Name                 Code                                      =>  Evaluation          =>  Normal form
  0  Instantiation (1:1)
 12  Output (12:19)       output( 'Please enter your ID.' )
 13  Input (13:19)        input( userID )                           =>  input( userID?11 )
 28  Output (20:21)       output( 'Please enter your PIN.' )
 39  Input (26:19)        input( key )                              =>  input( key?12 )
 39  Input (26:19)        input( key )                              =>  input( key?13 )
 39  Input (26:19)        input( key )                              =>  input( key?14 )
 72  Output (44:23)       output( 'PIN is incorrect. Please retry.' )
[Tests/access-control.sdl]$ path.conditions
No.  Name                  Code                  =>  Evaluation                          =>  Normal form
 14  Decision (15:21)      userID = 1            =>  userID?11 = 1
 53  Decision (33:21)      key >= 0 and key <= 9  =>  key?12 >= 0 and key?12 <= 9        =>  not ( 0 > key?12 ) and not ( key?12 > 9 )
 53  Decision (33:21)      key >= 0 and key <= 9  =>  key?13 >= 0 and key?13 <= 9        =>  not ( 0 > key?13 ) and not ( key?13 > 9 )
 53  Decision (33:21)      key >= 0 and key <= 9  =>  key?14 >= 0 and key?14 <= 9        =>  not ( 0 > key?14 ) and not ( key?14 > 9 )
 66  If statement (37:24)  not ( pin = pinOK )    =>  not ( 100 * key?12 + 10 * key?13 +
                                                        key?14 = 209 )
[Tests/access-control.sdl]$ externals.print ⇐print all external variables along with a valid assignment
Name        Value
userID?11 = [1]
key?12    = [0]
key?13    = [0]
key?14    = [0]
[Tests/access-control.sdl]$ externals.select max ⇐choose maximum values for external variables
[Tests/access-control.sdl]$ externals.print ⇐print another valid assignment
Name        Value
userID?11 = [1]
key?12    = [9]
key?13    = [9]
key?14    = [9]
[Tests/access-control.sdl]$ quit
Goodbye!
```

10.4 Discussion

Combining Symbolic Execution with State Space Reduction Techniques. An open issue is the combination of symbolic execution with some of the state space reduction techniques mentioned in chapter 8. Implementing a partial order simulation should be no problem as long as the algorithm does not depend on variables values. The *Independence Prioritizing Simulation* and the *Condition Locking Simulation* presented by Toggweiler (1995, chapter 11) solely consider control flow and signal flow aspects and thus can be used together with symbolic execution techniques.

On the other hand, the Supertrace algorithm is based on comparing global system states. In the context of symbolic execution, states are not only characterized by symbolic variables values but also by path conditions. To compare two states, both variable values and path conditions have to be transformed into a canonical form. Thereby, it is not sufficient to consider them in isolation. Instead, global operations such as relabeling external variables are needed. In addition, those path conditions which only contain outdated external variables must be filtered out. Outdated external variables are variables which do not occur anymore in the term of any variable and hence do not contribute to the current system state.

In principle, two identical symbolic states may have path conditions and variable values of totally different structure such that a substantial amount of normalization operations is needed to transform them into a canonical form and prove their equality. However, in distributed systems, identical states mainly occur due to the different *ordering* of events. (Another cause for identical states are loops.) In that case, path conditions and variable terms for two identical states do not differ or differ only slightly.

Therefore, some kind of *weak normalization* procedure is conceivable that allows to identify states in acceptable time but at the cost of false negatives. Such a weak normalization could be restricted to the normalization of individual variables, relabeling/renumbering of external variables throughout the terms of all system variables (variables are assumed to be in a fixed order) and reordering of path conditions (the exact order criteria is indifferent).

Another important aspect is the fact that two states may not only be equal or different — one state may also subsume another one. However, detecting such a relation requires a proof with *all*-quantified variables. A pragmatic solution for implementing the Supertrace heuristic is subject to further studies.

External Synonyms. A problem that has been identified within the VALIBOSE project concerns the handling of system parameters, i.e., SDL external synonyms, and uninitialized system variables when its comes to proving the feasibility of a path. In the existing prototype, external synonyms are treated like external variables, i.e., the constraint solver tries to find an assignment such that the path conditions are met.

However, for test generation purposes it is necessary to prove that the given path is executable for *any* value that a system parameter can take (because no assumption can

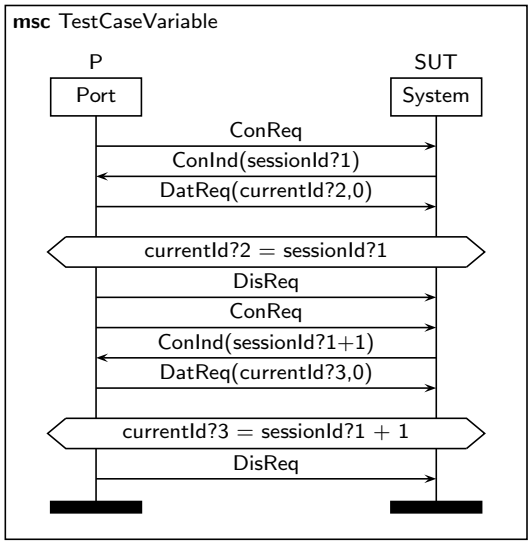


Figure 10.4: MSC *TestCaseVariable*

be made on the concrete value that is chosen for an implementation). This, of course, goes beyond the capabilities of existing CSP solvers, i.e., it can only be proven by enumeration.

Test Case Variables. The test case variable problem discussed in section 10.1 turned out to be difficult even with symbolic execution. For illustration purposes, see the MSC in figure 10.4. When a connection is established (*ConReq*/*ConInd*), the SUT returns a unique session id that must be used for subsequent data transmissions (*DatReq*). The relation between the value returned by *ConInd* and the first parameter of *DatReq* is clearly described in the MSC.

Nevertheless, an automatic tool fails to produce correct TTCN code for such an MSC for two reasons: First, the variable term in an output event may not only consist of a single external variable that can be mapped directly to a test case variable in TTCN (compare with *ConInd(sessionId?1+1)*). Second, the relations between output and input values are expressed by the path condition. These path conditions may become arbitrarily complex and also involve other relations than equality.

As a compromise, a test generation tool may produce TTCN stubs with annotations that have to be resolved by the test specifier. A TTCN fragment for the MSC discussed above is shown in figure 10.5(a), a revised test case produced by manual modifications is given in figure 10.5(b).

General Limitations of Symbolic Execution. There are several limitations of symbolic execution that may restrict its application. For example, symbolic execution is only possible if the source code of the complete program or specification is available. If external libraries are used, an interface specification in terms of input and output behavior is

```
p.send( ConReq : {} );
p.receive( ConInd : { * /*sessionId?1*/ } ) -> testCaseVar;
p.send( DatReq : { ??? /*currentId?2 ; currentId?2 = sessionId?1*/, 0 } );
p.send( DisReq : {} );
p.send( ConReq : {} );
p.receive( ConInd : { * /*sessionId?1 + 1*/ } ) -> testCaseVar;
p.send( DatReq : { ??? /*currentId?3 ; currentId?3 = sessionId?1 + 1*/, 0 } );
p.send( DisReq : {} );
```

(a) TTCN stub with annotations

```
p.send( ConReq : {} );
p.receive( ConInd : { * } ) -> testCaseVar;
p.send( DatReq : { testCaseVar, 0 } );
p.send( DisReq : {} );
p.send( ConReq : {} );
p.receive( ConInd : { * } ) -> testCaseVar;
p.send( DatReq : { testCaseVar, 0 } );
p.send( DisReq : {} );
```

(b) Revised TTCN code

Figure 10.5: Code generation for MSC *TestCaseVariable*

needed. With regard to test generation based on SDL, it can be assumed that an SDL specification is complete such that this problem does not occur.

Another problem concerns the access to array elements. Since variable values are described by symbolic terms, the array index might not be unique at the time of execution. Therefore, a choice point with – in worst case – as many alternatives as the size of the array has to be introduced. A pragmatic solution to circumvent this problem is to instantiate the index variable in such a case, i.e., to partially switch back from symbolic execution to "regular execution" with concrete values.

Despite the open issues discussed above, the author believes that symbolic execution is a powerful technique that can cope with the most challenging problems of automatic test generation. The ValiBOSE tool is subject to further developments. In particular, it is planned to support a wide range of SDL language concepts.

11 Conclusions

In this thesis, the automatic generation of abstract test cases based on formal specifications has been discussed. Solutions have been proposed for various problem areas, ranging from the generation of test cases for distributed test architectures, to efficient state space exploration and user-friendly representation of test cases. In addition, suggestions for possible future improvements have been made.

The applicability of automatic test generation has been demonstrated by AUTOLINK, a commercial tool that was developed in cooperation with TELELOGIC AB. AUTOLINK allows to generate conformance test suites in TTCN-2 format based on SDL specifications and MSC test purposes.

Case studies at ETSI have shown that automatic test generation can save both time and costs. However, there are two key factors which decide about the applicability and success of automatic test generation:

- The availability of a detailed, formal specification.
- The availability of a powerful and user-friendly test generation tool.

Formal specifications are mainly used by international standardization organizations such as ETSI or ITU-T and by major telecommunication companies. For test generation purposes, a formal specification should cover most aspects of the corresponding protocol, i.e., it should be as close to a concrete implementation as possible.

Concerning tool support, a practical test generation tool must be comprehensive. The lack of a single feature, e.g., support for test suite parameters, can mean a significant amount of additional work in terms of manual post-processing. Beyond that, it can jeopardize the overall applicability of the tool. Moreover, it is essential that test generation is integrated into the complete development process and a continuous tool chain is available for this process.

Some test generation techniques require adjustments by the test specifier. This, of course, presupposes a good knowledge of the underlying algorithms and principles. Feature interactions make these adjustments even more difficult. A typical example is the choice of a suitable set of heuristics if a complete state space exploration is impossible. Even experienced users have difficulties in predicting the consequences of each parameter in combination with other factors. A user-friendly solution should hide its technology and adapt itself automatically to a concrete task.

Participation in the AUTOLINK project and in the development of various test suites at ETSI has shown that there is a large gap between the theoretical concepts of scientists

on the one hand and the practical work of engineers on the other hand. There are several reasons why research results have difficulties in finding their way into practice:

- Many sophisticated methods can only be applied to rather small examples. For instance, fault model-based test generation and the computation of UIO sequences are feasible for Inres but they are rather unpromising undertakings for protocols of the size of Core INAP CS-2.
- Theoretical work is based on simplified models and abstracts from implementation details. In contrast, practical systems are very complex and full of constraints. A test generation tool for SDL that does not support a wide range of language constructs is practically irrelevant. For this reason, the correct implementation of advanced techniques such as partial order simulation can become arbitrarily complicated and economically unacceptable.
- Scientists and engineers tend to focus on different issues. Very often, the practitioner is struggling with problems that do not draw the attention of the academic world. In the context of automatic test generation, the readability of abstract test suites plays no vital role in academia but it is a major concern in standardization.

This thesis builds a bridge between both theory and practice, by providing pragmatic, scientific solutions for real-life problems.

A List of Abbreviations

ADT	Abstract Data Type
AN	Access Network
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitive
AST	Abstract Syntax Tree
ATM	Abstract Test Method / Asynchronous Transfer Mode
ATS	Abstract Test Suite
BBCC	Broadband Bearer Connection Control
BFS	Breadth-First Search
CEFSM	Communicating Extended FSM
CM	Coordination Message
CORBA	Common Object Request Broker Architecture
CP	Coordination Point
CSP	Constraint Satisfaction Problem
CT	Constraint Theory
CTL	Computation Tree Logic
CTMF	Conformance Testing Methodology and Framework
DCOM	Distributed Common Object Model
DFG	Data Flow Graph
DFS	Depth-First Search
DS	Distinguishing Sequence
EBNF	Extended Backus-Naur Form
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute
FD	Finite Domain
FDT	Formal Description Technique
FIFO	First In, First Out
FMCT	Formal Methods in Conformance Testing
FSM	Finite State Machine
HMSC	High-level MSC
ICS	Implementation Conformance Statement
IDL	Interface Definition Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
INAP CS-2	Intelligent Network Application Protocol, Capability Set 2

Inres	Initiator and Responder
IS	International Standard
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
IUT	Implementation Under Test
IXIT	Implementation eXtra Information for Testing
LCSAJ	Linear Code Sequence And Jump
LT	Lower Tester
LTCF	Lower Tester Control Function
LTS	Labeled Transition System
MOT	Means of Testing
MSC	Message Sequence Chart
MTC	Main Test Component
OSI	Open Systems Interconnection
PCO	Point of Control and Observation
PDU	Protocol Data Unit
PICS	Protocol Implementation Conformance Statement
PIXIT	Protocol Implementation eXtra Information for Testing
POSIX	Portable Operating System Interface for UNIX
PPO	Possible Pass Observable
PROLOG	Programming in Logic
PTC	Parallel Test Component
RTMC	Real Time Management Co-ordination
SAP	Service Access Point
SCS	System Conformance Statement
SDL	Specification and Description Language
SDU	Service Data Unit
SN	Service Node
STF	Specialist Task Force
SUT	System Under Test
TCP	Test Coordination Procedures
TR	Technical Report
TSS & TP	Test Suite Structure and Test Purposes
TTCN-2	Tree and Tabular Combined Notation 2
TTCN.GR	TTCN Graphical Form
TTCN.MP	TTCN Machine Processable Form
TTCN-3	Testing and Test Control Notation 3
TTCN-3 GFT	TTCN-3 Graphical Presentation Format
TTCN-3 TFT	TTCN-3 Tabular Presentation Format
UIO	Unique Input Output
UML	Unified Modeling Language
UPO	Unique Pass Observable
UT	Upper Tester
ValiBOSE	VALIdation Based On Symbolic Execution

VB5.1 / 5.2	Broadband “V” reference point 5.1 / 5.2
XSLT	Extensible Stylesheet Language Transformations

B TTCN-2 Test Suite for the Inres Protocol

Test Suite Overview

Test Suite Structure			
Suite Name : TestsForInres			
Standards Ref :			
PICS Ref :			
PIXIT Ref :			
Test Method(s) : Local test method			
Comments :			
Test Group Reference	**Selection Ref**	**Test Group Objective**	**Page Nr**
BasicInterconnectionTests/		Determine whether there is sufficient conformance for interconnection to be possible	210
BehaviorTests/		Determine the extent to which dynamic conformance requirements are met	210
Detailed Comments :			

Test Case Index				
Test Group Reference	**Test Case Id**	**Selection Ref**	**Description**	**Page Nr**
BasicInterconnectionTests/	SingleDataTransfer			210
BehaviorTests/	DataLoss	InopportuneEvents		210
Detailed Comments :				

Test Step Index			
Test Step Group Reference	**Test Step Id**	**Description**	**Page Nr**
	MediumAccess		211
Detailed Comments :			

Default Index			
Default Group Reference	**Default Id**	**Description**	**Page Nr**
Failures/	MTCFailure		211
Failures/	PTCFailure		211
Detailed Comments :			

Import Part

Imports			
Source Name : ServiceUser			
Source Ref :			
Standards Ref :			
Comments :			
Object Name	**Object Type**	**Source Name**	**Comments**
UserPDU	ASN1_PDU_TypeDef		
someUserPDU	ASN1_PDU_Constraint		
Detailed Comments :			

Declarations Part

ASN.1 Type Definition	
Type Name	: InresPDUType
Encoding Variation	:
Comments	:
Type Definition	
ENUMERATED { CR(1), CC(2), DR(3), DT(4), AK(5) }	
Detailed Comments :	

ASN.1 Type Definition	
Type Name	: SequenceNumber
Encoding Variation	:
Comments	:
Type Definition	
ENUMERATED { zero(0), one(1) }	
Detailed Comments :	

Encoding Definitions			
Encoding Rule Name	**Reference**	**Default**	**Comments**
BER_1997	ITU-T X.690 (12/97)	TRUE	Basic Encoding Rules
PER_BASIC_UNALIGNED_1997	ITU-T X.691 (12/97)		Packed Encoding Rules
Detailed Comments :			

Test Suite Parameter Declarations			
Parameter Name	**Type**	**PICS/PIXIT Ref**	**Comments**
maxRepetitions	INTEGER		
testInopportuneEvents	BOOLEAN		e.g., responder does not acknowledge data transfers
Detailed Comments :			

Test Case Selection Expression Definitions		
Expression Name	**Selection Expression**	**Comments**
InopportuneEvents	testInopportuneEvents = TRUE	
Detailed Comments :		

Test Suite Constant Declarations			
Constant Name	**Type**	**Value**	**Comments**
maxTestCaseTime	INTEGER	50	maximum execution time of a single test case
maxTransferTime	INTEGER	30	maximum execution time of a single data transfer
Detailed Comments :			

Test Case Variable Declarations			
Variable Name	**Type**	**Value**	**Comments**
receipt	INTEGER	0	
seqNumber	SequenceNumber		
Detailed Comments :			

PCO Type Declarations		
PCO Type	**Role**	**Comments**
InitiatorSAP	UT	
MediumSAP	LT	
Detailed Comments :		

PCO Declarations			
PCO Name	**PCO Type**	**Role**	**Comments**
ISAP1	InitiatorSAP	UT	
MSAP2	MediumSAP	LT	
Detailed Comments :			

Coordination Point Declarations	
CP Name	**Comments**
CoordinationPoint	Message exchange between MTC and PTC
Detailed Comments :	

Timer Declarations			
Timer Name	**Duration**	**Unit**	**Comments**
testCaseTimer	maxTestCaseTime	s	
supervisionTimer	maxTransferTime	s	
Detailed Comments :			

Test Component Declarations				
Component Name	**Component Role**	**Nr PCOs**	**Nr CPs**	**Comments**
MainTC	MTC	1	1	
ParallelTC	PTC	1	1	
Detailed Comments :				

Test Component Configuration Declaration			
Configuration Name : StandardConfiguration			
Comments :			
Components Used	**PCOs Used**	**CPs Used**	**Comments**
MainTC	ISAP1	CoordinationPoint	
ParallelTC	MSAP2	CoordinationPoint	
Detailed Comments :			

ASN.1 ASP Type Definition
ASP Name : ICONreq
PCO Type : InitiatorSAP
Comments :
Type Definition
SEQUENCE {}
Detailed Comments :

ASN.1 ASP Type Definition
ASP Name : ICONconf **PCO Type** : InitiatorSAP **Comments** :
Type Definition
SEQUENCE {}
Detailed Comments :

ASN.1 ASP Type Definition
ASP Name : IDATreq **PCO Type** : InitiatorSAP **Comments** :
Type Definition
SEQUENCE { iSDU UserPDU }
Detailed Comments : A User PDU on layer n+1 becomes an Inres SDU on layer n

ASN.1 ASP Type Definition
ASP Name : IDISreq **PCO Type** : InitiatorSAP **Comments** :
Type Definition
SEQUENCE {}
Detailed Comments :

ASN.1 ASP Type Definition
ASP Name : IDISind **PCO Type** : InitiatorSAP **Comments** :
Type Definition
SEQUENCE {}
Detailed Comments :

ASN.1 ASP Type Definition
ASP Name : MDATreq **PCO Type** : MediumSAP **Comments** :
Type Definition
SEQUENCE { mSDU InresPDU }
Detailed Comments : An Inres PDU on layer n becomes a Medium SDU on layer n-1

ASN.1 ASP Type Definition
ASP Name : MDATind **PCO Type** : MediumSAP **Comments** :
Type Definition
SEQUENCE { mSDU InresPDU }
Detailed Comments : An Inres PDU on layer n becomes a Medium SDU on layer n-1

ASN.1 PDU Type Definition	
PDU Name	: InresPDU
PCO Type	:
Encoding Rule Name	: PER_BASIC_UNALIGNED_1997
Encoding Variation	:
Comments	: Apply Packed Encoding Rules
Type Definition	

SEQUENCE {
iPDUType InresPDUType,
seqNo SequenceNumber OPTIONAL,
iSDU UserPDU OPTIONAL
}	

Detailed Comments : A User PDU on layer n+1 becomes an Inres SDU on layer n	

ASN.1 CM Type Definition	
CM Name	: Notification
Comments	:
Type Definition	
SEQUENCE {}	
Detailed Comments :	

Constraints Part

ASN.1 ASP Constraint Declaration	
Constraint Name	: InresConnectionRequest
ASP Type	: ICONreq
Derivation Path	:
Comments	:
Constraint Value	
{}	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: InresConnectionConfirmation
ASP Type	: ICONconf
Derivation Path	:
Comments	:
Constraint Value	
{}	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: InresDataRequest(data : UserPDU)
ASP Type	: IDATreq
Derivation Path	:
Comments	:
Constraint Value	
{ iSDU data }	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: InresDisconnectionRequest
ASP Type	: IDISreq
Derivation Path	:
Comments	:
Constraint Value	
{}	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: InresDisconnectionIndication
ASP Type	: IDISind
Derivation Path	:
Comments	:
Constraint Value	
{}	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: ConnectionRequest
ASP Type	: MDATind
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU { iPDUType CR } }	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: MediumDataRequest(data : InresPDU)
ASP Type	: MDATreq
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU data }	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: DataTransfer(data : UserPDU)
ASP Type	: MDATind
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU { iPDUType DT, seqNo ?, iSDU data } }	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: DataAcknowledgment(number : SequenceNumber)
ASP Type	: MDATreq
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU { iPDUType AK, seqNo number } }	
Detailed Comments :	

ASN.1 ASP Constraint Declaration	
Constraint Name	: DisconnectionRequest
ASP Type	: MDATind
Derivation Path	:
Comments	:
Constraint Value	
{ mSDU { iPDUType DR } }	
Detailed Comments :	

ASN.1 PDU Constraint Declaration	
Constraint Name	: ConnectionConfirmation
PDU Type	: InresPDU
Derivation Path	:
Encoding Rule Name	:
Encoding Variation	:
Comments	: This constraint is used with constraint 'MediumDataRequest'
Constraint Value	
{ iPDUType CC }	
Detailed Comments :	

ASN.1 CM Constraint Declaration	
Constraint Name	: acknowledgmentSent
CM Type	: Notification
Derivation Path	:
Comments	:
Constraint Value	
{}	
Detailed Comments :	

Dynamic Part

Test Case Dynamic Behaviour					
Test Case Name : SingleDataTransfer					
Group : BasicInterconnectionTests/					
Purpose :					
Configuration : StandardConfiguration					
Default : MTCFailure					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		CREATE(ParallelTC:MediumAccess)			
2		START testCaseTimer			
3		+Preamble			
4		START supervisionTimer			restrict time of data transfer
5		ISAP1 ! IDATreq	InresDataRequest (someUserPDU)		data transfer
6		CoordinationPoint ? Notification	acknowledgmentSent	(PASS)	delay disconnection request until 'ptc' has received and acknowledged the data
7		CANCEL supervisionTimer			cancel timer to avoid a timeout in the following
8		+Postamble			
9		? DONE(ParallelTC)		R	
10		ISAP1 ? IDISind	InresDisconnection-Indication	INCONC	
		Preamble			
11		ISAP1 ! ICONreq	InresConnectionRequest		
12		ISAP1 ? ICONconf	InresConnection-Confirmation		
13		ISAP1 ? IDISind	InresDisconnection-Indication	INCONC	
		Postamble			
14		ISAP1 ! IDISreq	InresDisconnection-Request		
15		ISAP1 ? IDISind	InresDisconnection-Indication		
Detailed Comments :					

Test Case Dynamic Behaviour					
Test Case Name : DataLoss					
Group : BehaviorTests/					
Purpose :					
Configuration : StandardConfiguration					
Default : MTCFailure					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
Detailed Comments : This test case is left unspecified					

Test Step Dynamic Behaviour					
Test Step Name : MediumAccess					
Group :					
Objective :					
Default : PTCFailure					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		MSAP2 ? MDATind	ConnectionRequest		
2		(receipt := 1)			first (received) connection request of the initiator
3		MSAP2 ! MDATreq	MediumDataRequest(ConnectionConfirmation)		
4	Loop	MSAP2 ? MDATind (seqNumber := MDATind.mSDU.seqNo)	DataTransfer(someUserPDU)	(PASS)	
5		MSAP2 ! MDATreq	DataAcknowledgment (seqNumber)		
6		CoordinationPoint ! Notification	acknowledgmentSent		inform the main test component that the data have been received and acknowledged
7		MSAP2 ? MDATind	DisconnectionRequest	PASS	
8		MSAP2 ? MDATind	DataTransfer(someUserPDU)	INCONC	data acknowledgment got lost
9		MSAP2 ? MDATind	ConnectionRequest		
10		[receipt <= maxRepetitions]			connection confirmation got lost probably due to a malfunction of the medium; resend it
11		(receipt := receipt + 1)			
12		MSAP2 ! MDATreq	MediumDataRequest ({ iPDUType CC })		
13		-> Loop			
14		[receipt > maxRepetitions]		FAIL	even over an unreliable medium, the initiator shall not resend its request that often
Detailed Comments :					

Default Dynamic Behaviour					
Default Name : MTCFailure					
Group : Failures/					
Objective :					
Comments: :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		ISAP1 ? OTHERWISE		FAIL	
2		? TIMEOUT		FAIL	
Detailed Comments :					

Default Dynamic Behaviour					
Default Name : PTCFailure **Group** : Failures/ **Objective** : **Comments:** :					
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		MSAP2 ? OTHERWISE		FAIL	
Detailed Comments :					

C TTCN-3 Module for the Inres Protocol

```
1: /*
2:  * TTCN-3 module for the 'Inres' protocol
3:  *
4:  * Copyright (C) 2002 Michael Schmitt <Michael.Schmitt@teststep.org>
5:  */
6:
7: module TestsForInres( integer maxRepetitions, boolean testInopportuneEvents ) {
8:   import from ServiceUser language "ASN.1:1997" {
9:     type UserPDU;
10:    const someUserPDU;
11:  }
12:
13:  group BasicDefinitions {
14:    type UserPDU InresSDU;   // the PDU on layer n+1 becomes an SDU on layer n
15:
16:    type enumerated InresPDUType { CR(1), CC(2), DR(3), DT(4), AK(5) };
17:
18:    type enumerated SequenceNumber { zero(0), one(1) };
19:
20:    type record InresPDU {
21:      InresPDUType iPDUType,
22:      SequenceNumber seqNo optional,
23:      InresSDU iSDU optional
24:    }
25:
26:    type InresPDU MediumSDU;   // the PDU on layer n becomes an SDU on layer n-1
27:  } with { encode "PER-BASIC-UNALIGNED:1997" }
28:    // apply Packed Encoding Rules
29:
30:  const float maxTestCaseTime := 50;   // maximum execution time of a single test case
31:  const float maxTransferTime := 30;   // maximum execution time of a single data transfer
32:
33:  group CommunicationWithInitiator {
34:    type record ICONreq {};
35:    type record ICONconf {};
36:    type record IDATreq { InresSDU iSDU };
37:    type record IDISreq {};
38:    type record IDISind {};
39:
```

```
40:       type port InitiatorSAP message {
41:         out ICONreq, IDATreq, IDISreq;   // sent to SUT
42:         in ICONconf, IDISind;   // received from SUT
43:       }
44:     }
45:
46:     group CommunicationWithMedium {
47:       type record MDATreq { MediumSDU mSDU };
48:       type record MDATind { MediumSDU mSDU };
49:
50:       type port MediumSAP message {
51:         in MDATind;   // received from SUT
52:         out MDATreq;   // sent to SUT
53:       }
54:     }
55:
56:     group CommunicationBetweenTestComponents {
57:       signature acknowledgmentSent();
58:
59:       type port PortAtMTC procedure {
60:         in acknowledgmentSent;
61:       }
62:
63:       type port PortAtPTC procedure {
64:         out acknowledgmentSent;
65:       }
66:     }
67:
68:     group ComponentDefinitions {
69:       type component MainTC {
70:         port InitiatorSAP ISAP1;
71:         port PortAtMTC CoordinationPTC;
72:         timer supervisionTimer;
73:       }
74:
75:       type component ParallelTC {
76:         port MediumSAP MSAP2;
77:         port PortAtPTC CoordinationMTC;
78:       }
79:
80:       type component TestSystem {
81:         port InitiatorSAP ISAP1;
82:         port MediumSAP MSAP2;
83:       }
84:     }
85:
86:     group TemplateDefinitions {
87:       template IDATreq InresDataRequest( InresSDU data ) := {
88:         iSDU := data
89:       }
```

```
90:
91:       template MDATind ConnectionRequest := {
92:         mSDU := { iPDUType := CR, seqNo := omit, iSDU := omit }
93:       }
94:
95:       template MediumSDU ConnectionConfirmation := {   // this template is used with
96:         iPDUType := CC, seqNo := omit, iSDU := omit   // template 'MediumDataRequest'
97:       }
98:
99:       template MDATreq MediumDataRequest( template MediumSDU data ) := {
100:        mSDU := data
101:      }
102:
103:      template MDATind DataTransfer( InresSDU data ) := {
104:        mSDU := { iPDUType := DT, seqNo := ?, iSDU := data }
105:      }
106:
107:      template MDATreq DataAcknowledgment( SequenceNumber number ) := {
108:        mSDU := { iPDUType := AK, seqNo := number, iSDU := omit }
109:      }
110:  }
111:
112:  altstep MTCFailure() runs on MainTC {
113:    [ ] ISAP1.receive {
114:          setverdict( fail );
115:          stop;
116:        }
117:    [ ] any timer.timeout {
118:          setverdict( fail );
119:          stop;
120:        }
121:  }
122:
123:  altstep PTCFailure() runs on ParallelTC {
124:    [ ] MSAP2.receive {
125:          setverdict( fail );
126:          stop;
127:        }
128:  }
129:
130:  altstep ReceptionIDISind( verdicttype result ) runs on MainTC {
131:    [ ] ISAP1.receive( IDISind : {} ) {
132:          setverdict( result );
133:          stop;
134:        }
135:  }
136:
```

```
137:    function MediumAccess() runs on ParallelTC {
138:      var integer   receipt;
139:      var default   def := activate( PTCFailure() );
140:      var MDATind   indication;
141:
142:      MSAP2.receive( ConnectionRequest );
143:      receipt := 1;   // first (received) connection request of the initiator
144:
145:      MSAP2.send( MediumDataRequest( ConnectionConfirmation ) );
146:
147:      alt {
148:        [ receipt <= maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
149:            // connection confirmation got lost probably due
150:            // to a malfunction of the medium; resend it
151:            receipt := receipt + 1;
152:            MSAP2.send( MediumDataRequest( { CC, omit, omit } ) );
153:            repeat;
154:          }
155:        [ receipt > maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
156:            // even over an unreliable medium, the initiator
157:            // shall not resend its requests that often
158:            setverdict( fail );
159:            stop;
160:          }
161:        [ ] MSAP2.receive( DataTransfer( someUserPDU ) ) -> value indication {
162:            /* empty */
163:          }
164:      }
165:
166:      MSAP2.send( DataAcknowledgment( indication.mSDU.seqNo ) );
167:
168:      // inform the main test component that the data have been received and acknowledged
169:      CoordinationMTC.call( acknowledgmentSent : {} );
170:      CoordinationMTC.getreply( acknowledgmentSent : {} );
171:
172:      alt {
173:        [ ] MSAP2.receive( MDATind : { mSDU := { DR, omit, omit } } ) {
174:            setverdict( pass );   // disconnection request
175:          }
176:        [ ] MSAP2.receive( DataTransfer( someUserPDU ) ) {   // data acknowledgment got lost
177:            setverdict( inconc );
178:          }
179:      }
180:    }
181:
182:    testcase SingleDataTransfer() runs on MainTC system TestSystem {
183:      var ParallelTC   ptc;
184:      var default   def1, def2;
185:
186:      ptc := ParallelTC.create;
```

```
187:
188:       map( self:ISAP1, system:ISAP1 );
189:       map( ptc:MSAP2, system:MSAP2 );
190:
191:       connect( self:CoordinationPTC, ptc:CoordinationMTC );
192:
193:       ptc.start( MediumAccess() );
194:
195:       def1 := activate( MTCFailure() );
196:       def2 := activate( ReceptionIDISind( inconc ) );
197:
198:       ISAP1.send( ICONreq : {} );   // connection request
199:       ISAP1.receive( ICONconf : {} );   // connection confirmation
200:
201:       supervisionTimer.start( maxTransferTime );   // restrict time of data transfer
202:
203:       ISAP1.send( InresDataRequest( someUserPDU ) );   // data transfer
204:
205:       // delay disconnection request until 'ptc' has received and acknowledged the data
206:       CoordinationPTC.getcall( acknowledgmentSent : {} );
207:       CoordinationPTC.reply( acknowledgmentSent : {} );
208:
209:       supervisionTimer.stop;   // cancel timer to avoid a timeout in the following
210:
211:       deactivate( def2 );   // a disconnection indication is no undesirable event any longer
212:
213:       ISAP1.send( IDISreq : {} );   // disconnection request
214:       ISAP1.receive( IDISind : {} );   // disconnection indication
215:
216:       all component.done;
217:
218:       setverdict( pass );
219:    }
220:
221:    testcase DataLoss() runs on MainTC system TestSystem {
222:       // ...
223:    }
224:
225:    control {
226:       var verdicttype   overallVerdict := pass;
227:
228:       overallVerdict := execute( SingleDataTransfer(), maxTestCaseTime );
229:
230:       if ( overallVerdict == pass and testInopportuneEvents == true ) {
231:          overallVerdict := execute( DataLoss() );
232:       }
233:    }
234: } with { encode "BER:1997" }   // apply Basic Encoding Rules by default
```

Bibliography

Balzert, H. (1998). *Lehrbuch der Software-Technik*, volume 2. Spektrum Akademie Verlag, 1st edition.

Barták, R. (1996). Constraint Programming: In Pursuit of the Holy Grail. *ACM Computing Surveys*, 28A(4).

Bourhfir, C., Dssouli, R., Aboulhamid, E., and Rico, N. (1997). Automatic executable test case generation for efsm specified protocols. In *Proceedings of the International Workshop on Testing of Communicating Systems*, pages 75–90. Chapman & Hall.

Boyer, R. S., Elspas, B., and Levitt, K. N. (1975). SELECT – A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York. ACM.

Bronstein, I. N., Semendjajew, K. A., Musiol, G., and Mühlig, H. (1999). *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, 4. edition.

Cavalli, A. R. and Anido, R. (1997). Verification and testing techniques based on the finite state machine model. Rapport de Recherche 97 09 02, Institut National des Télécommunications, Evry, France.

Cavalli, A. R., Lee, D., Rinderknecht, C., and Zaïdi, F. (1999). Hit-or-Jump: An algorithm for embedded testing with applications to IN services. In *Proceedings of FORTE/PSTV*, pages 41–56.

Chow, T. S. (1978). Testing software design modeled by finite state machines. *IEEE-SE*, 4(3):178–187.

Clark, J. (1999). XSL Transformations (xslt) version 1.0. W3C Recommendation, `http://www.w3.org/TR/1999/REC-xslt-19991116`.

Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag.

Clocksin, W. F. and Mellish, C. S. (1994). *Programming in PROLOG*. Springer Verlag, 4th edition.

Coward, D. and Ince, D. (1995). *The Symbolic Execution of Software – The SYM-BOL system*. Computer Science: Research and Practice. Chapman & Hall, London.

Da Vinci Communications (2002). TTCN-3 tools. `http://www.davinci-communications.com/products_ttcn3.html`.

Dai, Z. R. (1999). TTCN Test Suite Generation with *Autolink* — Applied to a 3rd Generation Mobile Network Protocol. Diploma thesis, Medizinische Universität zu Lübeck, Lübeck, Germany.

Dai, Z. R., Grabowski, J., and Neukirchen, H. (2002). Timed TTCN-3 – A Real-Time Extension for TTCN-3. In *Testing Internet Technologies and Services. Proceedings of the IFIP TC6 14th International Conference on Testing of Communicating Systems (TestCom 2002)*, pages 407–424, Berlin.

De Nicola, R. and Hennessy, M. (1984). Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133.

Deraison, R. (2000). *The Nessus Attack Scripting Language Reference Guide*. `http://www.nessus.org/documentation.html`, 1.0.0pre2 edition.

Ebner, M., Yin, A., and Li, M. (2002). Definition and Utilisation of OMG IDL to TTCN-3 Mappings. In Schieferdecker, I., König, H., and Wolisz, A., editors, *TESTING OF COMMUNICATING SYSTEMS XIV — Application to Internet Technologies and Services*, pages 443–458. IFIP, Kluwer Academic Publishers.

Eertink, E. H. (1994). *Simulation Techniques for the Validation of LOTOS Specifications*. PhD thesis, University of Twente.

Ek, A., Grabowski, J., Hogrefe, D., Jerome, R., Koch, B., and Schmitt, M. (1997). Towards the Industrial Use of Validation Techniques and Automatic Test Generation Methods for SDL Specifications. In Cavalli, A. and Sarma, A., editors, *SDL '97 Time for Testing. SDL, MSC and Trends — Proceedings of the Eighth SDL Forum*, pages 245–259, Evry, France. Elsevier.

ETSI, European Telecommunications Standards Institute (1994). Technical Report 141 — Methods for Testing and Specification (MTS): Protocol and profile conformance testing specifications; The Tree and Tabular Combined Notation (TTCN) style guide. ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (1999a). DEN/SPAN-09047-1 / EN 301 217-1 V1.2.2 — V interfaces at the digital Service Node (SN); Interfaces at the VB5.2 reference point for the support of broadband or combined narrowband and broadband Access Networks (ANs); Part 1: Interface Specification. ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (1999b). DEN/SPS-03038-1 / EN 301 140-1 V1.3.4 — Intelligent Network (IN); Intelligent Network Application Protocol (INAP); Capability Set 2 (CS2); Part 1: Protocol Specification. ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2000a). DEN/SPS-03038-3 / EN 301 140-3 V1.1.3 — Intelligent Network (IN); Intelligent Network Application Protocol (INAP); Capability Set 2 (CS2); Part 3: Test Suite Structure and Test Purposes (TSS&TP) specification for Service Switching Function (SSF). ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2000b). DEN/SPS-03038-4 / EN 301 140-4 V1.1.3 — Intelligent Network (IN); Intelligent Network Application Protocol (INAP); Capability Set 2 (CS2); Part 4: Abstract Test Suite (ATS) specification and Partial Protocol Implementation eXtra Information for Testing (PIXIT) proforma for Service Switching Function (SSF). ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2000c). DEN/SPS-09046-3 / EN 301 005-3 V1.1.2 — V interfaces at the digital Service Node (SN); Interfaces at the VB5.1 reference point for the support of broadband or combined narrowband and broadband Access Networks (ANs); Part 3: Test Suite Structure and Test Purposes (TSS&TP) specification. ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2000d). DEN/SPS-09046-4 / EN 301 005-4 V1.1.2 — V interfaces at the digital Service Node (SN); Interfaces at the VB5.1 reference point for the support of broadband or combined narrowband and broadband Access Networks (ANs); Part 4: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT) proforma specification. ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2001a). DEN/SPAN-09047-3 / EN 301 217-3 V1.1.1 — V interfaces at the digital Service Node (SN); Interfaces at the VB5.2 reference point for the support of broadband or combined narrowband and broadband Access Networks (ANs); Part 3: Test Suite Structure and Test Purposes (TSS&TP). ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2001b). DEN/SPAN-09047-4 / EN 301 217-4 V1.1.1 — V interfaces at the digital Service Node (SN); Interfaces at the VB5.2 reference point for the support of broadband or combined narrowband and broadband Access Networks (ANs); Part 4: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2002a). ES 201 873-1 V2.2.1 — Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2002b). ES 201 873-2 V2.2.1 — Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT). ETSI, Sophia Antipolis, France.

ETSI, European Telecommunications Standards Institute (2002c). TR 101 873-3 V1.1.2 — Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT). ETSI, Sophia Antipolis, France.

Feijs, L. M. G., Goga, N., and Mauw, S. (2000). Probabilities in the TorX test derivation algorithm. In Graf, S., Jard, C., and Lahav, Y., editors, *Proceedings of the Second Workshop on SDL and MSC*, pages 173–188, Col de Porte, Grenoble, France.

Frühwirth, T. and Abdennadher, S. (1997). *Constraint-Programmierung: Grundlagen und Anwendungen*. Springer Verlag, Berlin, Heidelberg.

Gilly, D. (1994). *UNIX in a Nutshell — System V Edition*. O'Reilly & Associates, Sebastopol, California, 2nd edition.

Girgis, M. R. and Woodward, M. R. (1986). An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. In *Proceedings of the Workshop on Software-Testing*, pages 64–73, Banff.

Grabowski, J., Hogrefe, D., and Nahm, R. (1993). Test Case Generation with Test Purpose Specification by MSCs. In Færgemand, O. and Sarma, A., editors, *SDL '93 Using Objects. — Proceedings of the Sixth SDL Forum*. North-Holland.

Grabowski, J., Hogrefe, D., Nussbaumer, I., and Spichiger, A. (1995). Test Case Specification Based on MSCs and ASN.1. In *Proceedings of the Seventh SDL Forum*, Oslo, Norway. Elsevier.

Grabowski, J., Koch, B., Schmitt, M., and Hogrefe, D. (1999). SDL and MSC Based Test Generation for Distributed Test Architectures. In Dssouli, R., von Bochmann, G., and Lahav, Y., editors, *SDL '99 The next Millenium — Proceedings of the Nineth SDL Forum*, pages 389–404, Montreal, Canada. Elsevier.

Grabowski, J., Rudolph, E., and Schmitt, M. (2001). Die Spezifikationssprachen MSC und SDL — Teil 1: Message Sequence Chart (MSC). ***at*** — *Automatisierungstechnik*, 49(12):A19–A22.

Grabowski, J., Rudolph, E., and Schmitt, M. (2002). Die Spezifikationssprachen MSC und SDL — Teil 2: Specification and Description Language (SDL). ***at*** — *Automatisierungstechnik*, 50(2):A1–A4.

Grabowski, J., Scheurer, R., Toggweiler, D., and Hogrefe, D. (1996). Dealing with the complexity of state space exploration algorithms for SDL systems. In *Arbeitsberichte des Instituts für mathematische Maschinen- und Datenverarbeitung (Mathematik),Proceedings of the sixth GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems*, volume 20, Erlangen. University of Erlangen.

Grabowski, J. and Schmitt, M. (2002). TTCN-3 — Eine Sprache für die Spezifikation und Implementierung von Testfällen. ***at*** — *Automatisierungstechnik*, 50(3):A5–A8.

Graham, R. L., Knuth, D. E., and Patashnik, O. (1994). *Concrete Mathematics : A Foundation for Computer Science*. Addison–Wesley, 2nd edition.

Groz, R. and Risser, N. (1997). Eight years of experience in test generation from FDTs using TVEDA. In Mizuno, T., Shiratori, N., Higashino, T., and Togashi, A., editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, pages 465–480. IFIP, Chapman & Hall.

Hantler, S. L. and King, J. C. (1976). An Introduction to Proving the Correctness of Programs. *ACM Computing Surveys*, 8(3):331–353.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.

Hogrefe, D. (1989). *Estelle, LOTOS und SDL — Standard-Spezifikationssprachen für verteilte Systeme*. Springer Verlag, Berlin, Heidelberg, New York.

Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*. Prentice Hall.

IEEE, Institute of Electrical and Electronics Engineers (1990). IEEE Standard 610.12 — Standard Glossary of Software Engineering Terminology. IEEE, New York.

ILOG (1999a). *ILOG Solver 4.4 – Reference Manual*. ILOG S.A., France.

ILOG (1999b). *ILOG Solver 4.4 – User's Manual*. ILOG S.A., France.

ISO, International Organization for Standardization and IEC, International electrotechnical commission (1993). International Standard 10646, second edition: Information Technology – Universal Multiple Octet-Coded Character Set (UCS). ISO/IEC, Geneva, Switzerland.

ISO, International Organization for Standardization and IEC, International electrotechnical commission (1994a). International Standard 13210: Information Technology – Test methods for measuring conformance to POSIX (ANSI/IEEE Standard 1003.3-1991). ISO/IEC, Geneva, Switzerland.

ISO, International Organization for Standardization and IEC, International electrotechnical commission (1994b). International Standard 9646-1: Information technology – Open systems interconnection – Conformance testing methodology and framework, part 1: General concepts, Second Edition. ISO/IEC, Geneva, Switzerland.

ISO, International Organization for Standardization and IEC, International electrotechnical commission (1997). International Standard 9646-3: Information technology – Open systems interconnection – Conformance testing methodology and framework, part 3: Tree and Tabular Combined Notation, Second Edition. ISO/IEC, Geneva, Switzerland.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1992). Recommendation Z.100 — CCITT Specification and Description Language (SDL). ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1996a). Addendum 1 (10/96) to Recommendation Z.100 — CCITT Specification and Description Language (SDL). ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1996b). Recommendation Z.120 (10/96) — Message Sequence Chart (MSC). ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1997a). Recommendation X.680 (12/97) — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1997b). Recommendation Z.500 (5/97) — Framework on formal methods in conformance testing. ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1999a). Recommendation Z.100 (11/99) — Specification and Description Language (SDL). ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1999b). Recommendation Z.105 (11/99) — SDL combined with ASN.1 modules (SDL/ASN.1). ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1999c). Recommendation Z.107 (11/99) — SDL with embedded ASN.1. ITU, Geneva.

ITU-T, International Telecommunication Union — Telecommunication Standardization Sector (1999d). Recommendation Z.120 (11/99) — Message Sequence Chart (MSC). ITU, Geneva.

Kerbrat, A., Jéron, T., and Groz, R. (1999). Automated test generation from SDL specifications. In Dssouli, R., von Bochmann, G., and Lahav, Y., editors, *SDL '99 The next Millenium — Proceedings of the Nineth SDL Forum*, pages 135–151, Montreal, Canada. Elsevier.

Kneuper, R. (1992). Validation und Verifikation von Software durch symbolische Ausführung. In Liggesmeyer, P., Sneed, H. M., and Spillner, A., editors, *Testen, Analysieren und Verifizieren von Software*, Informatik Aktuell. Springer Verlag, Berlin.

Koch, B. (2001). *Test-purpose-based Test Generation for Distributed Test Architectures*. Inauguraldissertation, Medizinische Universität zu Lübeck, Lübeck, Germany.

Koch, B., Grabowski, J., Hogrefe, D., and Schmitt, M. (1998). Autolink — A Tool for Automatic Test Generation from SDL Specifications. In *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*, Boca Raton, Florida.

Kohavi, Z. (1978). *Switching and Finite Automata Theory*. McGraw-Hill, New York.

Kuikka, E. and Penttonen, M. (1995). Transformation of structured documents. Technical Report CS-95-46, Department of Computer Science, University of Waterloo, Waterloo, Canada.

Liggesmeyer, P. (1990). *Modultest und Modulverifikation – State of the Art*. BI-Wissenschaftsverlag, Mannheim.

Mayer, S. (2000). Automatic test generation for the Test Synchronization Protocol 1 with Autolink. Term paper, Medizinische Universität zu Lübeck, Lübeck, Germany.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Moesch, F. (2001). Entwurf und Implementation einer benutzerfreundlichen Eingabesprache für einen Testautomaten. Term paper, Medizinische Universität zu Lübeck, Lübeck, Germany.

Nahm, R. (1995). *Conformance Testing based on Formal Description Techniques and Message Sequence Charts*. PhD thesis, Institut für Informatik, University of Berne, Berne, Swiss.

Naito, S. and Tsunoyama, M. (1981). Fault detection for sequential machines by transition tours. In *Proceedings of the IEEE Fault Tolerant Computing Conference*, pages 238–243.

Object Management Group (1999). Unified modeling language specification; version 1.3. `http://www.rational.com/uml`.

Peng, W. W. and Wallace, D. R. (1993). Software error analysis. NIST Special Publication 500–209, National Institute of Standards and Technology (NIST), U.S. Department of Commerce, Gaithersburg, MD 20899.

Prinz, A., Eschbach, R., and Gotzhein, R. (2000). An Executable Formal Semantics for SDL-2000. In *Proceedings of the 2nd Workshop on SDL and MSC (SAM 2000)*, pages 249–261, Col de Porte, Grenoble. VERIMAG, IRISA, and SDL Forum Society.

Rauhamaa, M. (1990). A comparative study of methods for efficient reachability analysis. Series A: Research Reports No. 14, Department of Computer Science, Helsinki University of Technology, Espoo, Finland.

Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison–Wesley, Reading, Massachusetts.

Sabnani, K. K. and Dahbura, A. T. (1988). A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297.

Savoye, R. (2001). *DejaGNU – The GNU Testing Framework*. Free Software Foundation, 1.4.1 revision 0.6.1 edition.

Scheurer, R. (1997). *Demonstrating the Applicability of Automatic Test Case Generation Methods*. Inauguraldissertation, Institut für Informatik und angewandte Mathematik, Universität Bern, Bern.

Schmitt, M. (2000). The Development of a Parser for SDL-2000. Schriftenreihe der Institute für Informatik\Mathematik Report A-00-10, Medizinische Universität zu Lübeck, Lübeck, Germany.

Schmitt, M., Ebner, M., and Grabowski, J. (2000). Test Generation with AUTOLINK and TEST-COMPOSER. In Graf, S., Jard, C., and Lahav, Y., editors, *Proceedings of the Second Workshop on SDL and MSC*, pages 218–232, Col de Porte, Grenoble, France.

Schmitt, M., Ek, A., Grabowski, J., Hogrefe, D., and Koch, B. (1998). Autolink — Putting SDL-based test generation into practice. In Petrenko, A. and Yevtushenko, N., editors, *Testing of Communicating Systems. Proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS'98)*, volume 11, pages 227–243, Tomsk, Russia. Kluwer Academic Publishers.

Schmitt, M. and Koch, B. (2001). AUTOLINK *User Manual*. Telelogic AB, Malmö, Sweden.

Schmitt, M., Koch, B., Grabowski, J., and Hogrefe, D. (1997). Autolink — A Tool for the Automatic and Semi-Automatic Test Generation. In Wolisz, A., Schieferdecker, I., and Rennoch, A., editors, *Proceedings of the Seventh GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems*, pages 333–341, Berlin, Germany. GMD-Forschungszentrum Informationstechnik GmbH.

Schmitt, M., Koch, R., and Grabowski, J. (2001). A parser for TTCN-3. `http://www.itm.mu-luebeck.de/research/`.

Schöning, U. (1989). *Logik für Informatiker*, volume 56 of *Reihe Informatik*. BI-Wissenschaftsverlag, Mannheim, Wien, Zürich, 2nd edition.

SDL Forum Society (2002a). `http://www.sdl-forum.org`.

SDL Forum Society (2002b). SDL and MSC Tools. `http://www.sdl-forum.org/Tools/index.htm`.

Telelogic (2001). TAU *4.1 User's Manual. Chapter 53: The SDL Validator*. Telelogic, Malmö, Sweden.

Telelogic (2002a). TAU product description. `http://www.telelogic.com/products/tau/`.

Telelogic (2002b). TTCN-3 toolset. `http://www.telelogic.com/products/tau/languages/ttcn.cfm`.

Telelogic (formerly Verilog) (2002). OBJECTGEODE product description. `http://www.telelogic.com/products/additional/objectgeode/index.cfm`.

Testing Technologies (2002). TT Tool Series. `http://www.testingtech.com/products/TTToolSeries.html`.

Toggweiler, D. (1995). *Efficient Test Case Generation for distributed Systems specified by Automata*. Inauguraldissertation, University of Berne, Bern.

Touag, A. and Rouger, A. (1999). Methods and Methodology for an Incremental Test Generation from SDL Specifications. In Dssouli, R., von Bochmann, G., and Lahav, Y., editors, *SDL '99 The next Millenium — Proceedings of the Nineth SDL Forum*, pages 153–168, Montreal, Canada. Elsevier.

Tsang, E. (1996). *FOUNDATIONS OF CONSTRAINT SATISFACTION*. Department of Computer Science, University of Essex, Colchester, Essex, UK.

van Glabbeek, R. J. (1993). The Linear Time-Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves. In *Proceedings of the Fourth International Conference on Concurrency Theory (CONCUR '93)*, number 715 in Lecture Notes in Computer Science, pages 66–81. Springer Verlag.

Vasilevskii, M. P. (1973). Failure diagnosis of automata. *Kibernetika*, 4:98–108. Original in Russian.

Walter, T., Schieferdecker, I., and Grabowski, J. (1998). Test Architectures for Distributed Systems — State of the Art and Beyond. In Petrenko, A. and Yevtushenko, N., editors, *Proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS'98)*, volume 11, pages 149–174, Tomsk, Russia. Kluwer Academic Publishers.

West, C. H. (1992). Protocol validation - principles and applications. *Computer Networks and ISDN Systems*, 24:219–242. North-Holland.

Zhou, N.-F. (2000). *B-Prolog User's Manual*. Department of Computer and Information Science, Brooklyn College, The City University of New York, New York, USA, 4.0 edition.

Zieren, J. (2000). Automatische Generierung normativer Benutzermodelle aus SDL-Spezifikationen. Diploma thesis, Lehrstuhl für Technische Informatik, RWTH Aachen.

Curriculum Vitae
Michael Schmitt

Persönliche Daten

geboren am	14. Juli 1970 in Orsoy
Eltern	Joachim Schmitt, Diplom-Ingenieur, Ehefrau Elisabeth, geb. Huber
Familienstand	ledig
Staatsangehörigkeit	deutsch

Ausbildung / Beruflicher Werdegang

1977 – 1981	Grundschule in Neukirchen-Vluyn
1981 – 1990	Julius-Stursberg-Gymnasium in Neukirchen-Vluyn; Abschluss: Abitur
1990 – 1996	Informatik-Studium an der Universität Koblenz-Landau, Abteilung Koblenz, mit dem Anwendungsfach Computerlinguistik; Abschluss: Diplom mit Auszeichnung
Okt. 1996 – Sept. 2001	wissenschaftlicher Mitarbeiter am Institut für Telematik der Medizinischen Universität zu Lübeck
Okt. 2001 – März 2002	wissenschaftlicher Mitarbeiter am Institut für Telematik e.V., Trier
seit April 2002	wissenschaftlicher Mitarbeiter am Lehrstuhl von Prof. Meinel an der Universität Trier

---