UML-based Test Specification for Communication Systems

- A Methodology for the use of MSC and IDL in Testing -

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

vorgelegt von
Michael Ebner
aus Titisee-Neustadt

Göttingen 2004

Diese Dissertation ist elektronisch veröffentlicht und unter
http://webdoc.sub.gwdg.de/diss/2004/ebner/ebner.pdf archiviert.

This dissertation is published electronically and available via
http://webdoc.sub.gwdg.de/diss/2004/ebner/ebner.pdf.

D7

Referent:	Prof. Dr. Dieter Hogrefe
Korreferent:	Prof. Dr. Jens Grabowski
Tag der mündlichen Prüfung:	29. März 2004

Abstract

Nowadays, the complexity of modern telecommunication systems has increased significantly and the requirement for thorough and systematic testing is undisputed. The *Testing and Test Control Notation (version 3)* (TTCN-3) is an universal and standardised language for the specification and implementation of tests for communication systems. Many systems and in particular object-oriented systems are described using the *Unified Modeling Language* (UML). Therefore, UML models are an important source for test development and in particular for manual test purpose specification and automatic test generation. Thus, usage of UML from a test perspective is considered.

UML models provide interface information by class diagramms and description of scenarios by sequence diagrams respectively *Message Sequence Charts* (MSCs). Most UML tools permit conversion of class diagrams into *Interface Definition Language* (IDL) which widens applicability. The combination of TTCN-3 and UML by MSC and IDL is a new approach. Thus, new mappings for MSC and IDL to TTCN-3 have been worked out. Additionally, to widen usability and applicability, the deficiency of object-orientation in TTCN-3 is inspected and a proposal for an object-oriented revision is given.

Zusammenfassung

Die Komplexität moderner, verteilter Kommunikationssysteme hat sich erheblich gesteigert und die Notwendigkeit gründlichen und systematischen Testens ist unbestritten. Die *Testing and Test Control Notation (version 3)* (TTCN-3) ist eine universelle und standardisierte Sprache zur Spezifikation und Implementierung von Tests für verteilte Systeme. Viele Systeme, insbesondere objekt-orientierte Systeme, werden heutzutage mittels der *Unified Modeling Language* (UML) beschrieben und deshalb sind UML-Modelle eine wichtige Quelle für die Testentwicklung und insbesondere für die manuelle Testzweckspezifikation und die automatische Testgenerierung. Folglich wird die Verwendung von UML aus der Testperspektive heraus betrachtet.

UML-Modelle bieten Informationen über Schnittstellen durch Klassendiagramme und Szenarienbeschreibungen durch Sequenzdiagramme beziehungsweise *Message Sequence Charts* (MSCs) an. Die meisten UML-Werkzeuge erlauben die Konvertierung der Klassendiagramme in die *Interface Definition Language* (IDL), wodurch die Anwendbarkeit erweitert wird. Die Kombination von TTCN-3 und UML durch MSC und IDL ist ein neuer Ansatz. Deshalb wurden Abbildungen von MSC und IDL nach TTCN-3 ausgearbeitet. Um eine verbesserte Bedienbarkeit und Anwendbarkeit zu erreichen, wird zusätzlich die Verwendung von Objektorientierung in TTCN-3 untersucht und ein Vorschlag für eine objektorientierte Überarbeitung gegeben.

Acknowledgements

Firstly, I like to thank my supervisor Prof. Dr. Hogrefe for his kind support and the possibility to research under excellent conditions. The work at the Institute for Telematics in Lübeck and Telematics Group in Göttingen has been a very positive experience.

I would also like to thank my former colleagues in Lübeck and my current colleagues in Göttingen who have accompanied me through the years. It was always a pleasure to work with them. Special thanks go to Prof. Dr. Jens Grabowski, Helmut Neukirchen, and Dr. Michael Schmitt with whom I shared a lot of discussions and debates. It has been a very nice and interesting time.

This thesis would not be in its current shape without the comments of numerous people. Thus, I truly appreciate the efforts of Zhen Ru Dai, Helmut Neukirchen, Dr. Michael Schmitt, Rene Soltwisch, and Edith Werner.

I am also grateful to Prof. Dr. Jochen Seitz who has encouraged me to prepare a doctoral thesis. In addition, I like to thank Carmen and Barbara for their continuous motivation to finish this thesis.

Finally, I have to thank my parents for providing so much support for my unusual career path throughout the years.

Contents

1. Introduction *1*

2. Fundamentals of Testing *5*
2.1. Dynamic Testing Concepts 5
2.2. Problems of Object-Orientation for Testing 10
2.3. Test Generation 14
2.4. Testing and Test Control Notation 19
2.5. Summary 37

3. UML-based Testing *39*
3.1. Unified Modeling Language 39
3.2. Suitability of UML for Testing 41
3.3. UML-based Test Specification 43
3.4. Message Sequence Chart 45
3.5. Interface Definition Language 53
3.6. Summary and Outlook 58

4. Mapping of MSC to TTCN-3 *61*
4.1. Fundamental Concept 62
4.2. MSC Documents and Comments 64
4.3. Basic Message Sequence Charts 65
4.4. Structural Concepts 69
4.5. High-Level Message Sequence Charts 75
4.6. Summary and Outlook 76

5. Mapping of IDL to TTCN-3 *79*
5.1. Fundamental Concept 79
5.2. Lexical Conventions and Preprocessing 80
5.3. Structural Elements 82
5.4. Data Types 85
5.5. Communication Declaration 96
5.6. Names and Scoping 102
5.7. Summary and Outlook 103

6. Object-Oriented Enhancements for TTCN-3 *107*
6.1. Object-Orientation in TTCN-3 107
6.2. Object-Oriented Revision of TTCN-3 113
6.3. Summary and Outlook 116

7. Conclusion *119*

A. IDL Mapping Summary *121*
A.1. Conceptual IDL to TTCN-3 Mapping 121
A.2. Comparison of IDL, ASN.1, TTCN-2, and TTCN-3 Data Types 122
A.3. Examples 124

B. The TTCN-3 Inres Protocol Module *133*

Acronyms *139*

Bibliography *141*

List of Figures *151*

List of Tables *153*

1. Introduction

The complexity of modern telecommunication systems has increased significantly and the necessity for thorough and systematic testing is undisputed. For instance, conformance and functional testing is widely used in the telecommunication area. However, testing is an expensive and time-consuming task. Before concrete tests can be carried out on a system, much effort has to be spent on specifying what and how to test and on obtaining the test descriptions in a format that is accepted by the test equipment.

Testing of distributed systems based on Internet technologies has not matured that much. However, testing becomes more and more important if we consider the increasing amount of provided services and transfered data with Internet-based technologies. Furthermore, traditional telecommunication systems develop to Internet-based services to provide more powerful systems and to create new services. Testing of these new Internet-based applications is as crucial for their success as in the telecommunication area.

Testing has to be integrated into the development process. Therefore, to reduce testing effort tests should be generated from system specification which is named *Computer Aided Test Generation* (CATG). Manual test generation is error-prone wherefore test generation must be automated to be effective and repeatable. However, test generation based on state space exploration is helpful but generates frequently inefficient tests, lacks to cover specific parts, or may not be possible because of an incomplete or missing specification. Thus, *scenario-based, manual test specification* is interesting where the test designer can focus on specific elements in the *System Under Test* (SUT) and requires no test specific knowledge like the used test language.

In the telecommunication area, the *Tree and Tabular Combined Notation* (TTCN) is used as a standardised test description language. *Tree and Tabular Combined Notation (version 2)* (TTCN-2) (ISO/IEC 1998b) has been applied successfully to functional testing of communication protocols for years. *Testing and Test Control Notation (version 3)* (TTCN-3) has been especially designed to test CORBA-based systems to satisfy the demand for testing Internet-based distributed systems (ETSI 2002a). *Com-*

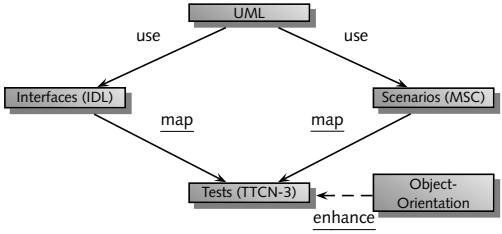


Figure 1.1.: *Fundamental concept of the thesis*

mon Object Request Broker Architecture (CORBA) is a standard architecture for distributed object systems and standardised by the *Object Management Group* (OMG) (OMG 2001b). Many systems and in particular object-oriented systems can be described using the *Unified Modeling Language* (UML) which is also defined by the OMG (OMG 2003c). Thus, UML models are an important source for test development and using UML from a test perspective has to be considered. Additionally, there is ongoing work on an UML *Testing Profile* (UTP) which can be mapped to TTCN-3.

Scope

UML provides sequence diagrams which are very similar to *Message Sequence Charts* (MSCs). Thus, they are used to specify test scenarios. Most UML tools permit generation of *Interface Definition Language* (IDL) wherefore it is used to provide interface information (see Figure 1.1). In addition, it widens application because IDL is very common. For instance, CORBA systems are using IDL to describe their object interfaces and there exist mappings to different languages like *Abstract Syntax Notation One* (ASN.1) and *Web Services Description Language* (WSDL). Using TTCN as test description language is quite natural because of its successful application in the telecommunication area by using automatic test generation, and the possible usage of TTCN-3 with UML via the UML *Testing Profile* (UTP). In addition, it was successfully applied for testing CORBA-based systems. The combination of TTCN-3, UML sequence diagrams substituted by MSCs, and IDL to provide *scenario-based, manual test specification* is a new approach. Thus, new mappings for IDL and MSC to TTCN-3 have to be worked out which can be seen in chapter 4 and chapter 5 (see Figure 1.1).

To widen usability and applicability of TTCN-3, the object-oriented concepts in TTCN-3 are inspected and a proposal for an object-oriented revision is made (see Figure 1.1).

Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces some fundamentals on testing with focus on object-orientation and test generation and then introduces the test description language TTCN-3. Chapter 3 discusses the usage of UML for testing of systems and explains the concept of scenario-based testing with UML. Furthermore, it introduces MSC and IDL. Chapter 4 deals with a mapping of MSCs to TTCN-3 to allow test generation based on scenarios defined in UML. In chapter 5, a mapping of IDL to TTCN-3 is detailed to use interface information for test generation. The deficiency of object-orientation in TTCN-3 and an object-oriented revision are discussed in chapter 6. Finally, conclusions are given.

2. Fundamentals of Testing

Testing is an important part of the analytical quality control of information technology systems and hence, is part of each software/hardware engineering process to assure functionality and reliability (Balzert 1998; Kaner et al. 1999). Test methods aim at detecting faults whereas *verification* methods try to show the formal correctness against the specification, and *validation* methods try to confirm the suitability of systems or system components for the application purpose.

There are static and dynamic test methods. Static methods like inspection, review, and walkthrough analyse the source code while dynamic methods execute the system and provide concrete input data (Myers 2001). Static testing methods are especially useful in early stages of programming and for finding structural problems.

Dynamic testing allows testing in the environment[1] of the system but cannot prove the absence of faults because the selected test input data does not cover all cases. Testing all cases would be the same like formal verification which mostly is too difficult for modern systems. However, when it comes to the later development stages, in which systems get larger and more complex, dynamic testing is a way to get a better coverage of the system as it can be done with static testing. Hence, a better confidence in the system is possible. In this thesis, only dynamic testing for distributed, object-oriented systems is considered.

The testing area is divided into many fields wherefore some classifications are given first. Afterwards, some remarks on problems in testing object-oriented systems and test generation are given. Furthermore, the test description language TTCN-3 gets explained. Finally, the chapter is summarised.

2.1. Dynamic Testing Concepts

The dynamic testing area is divided into many fields with different methods, procedures, and objectives depending on the application area. Some characteristics to classify dynamic testing are

[1] It is quite common to simulate the environment.

- the implementation type (target platform),
- the place in the development cycle,
- the knowledge about the underlying system,
- the test objective,
- the test data selection, and
- the test result authority.

Below, these characteristics are explained in more detail (Balzert 1998; Kaner et al. 1999; Myers 2001).

Type of Implementation

Testing depends on the target platform which can be *hardware*, *software*, or both. Hardware testing is done on the physical level with signal input and output. This concerns physical elements such as transistors, gates, and circuits, or functional elements like busses. Software testing is done on the logical layer where the hardware is assumed to be correct. In the following, we only consider software testing.

Development Cycle

Faults can occur in each phase of the software development cycle. Therefore, the phase influences the kind of test such as module test, integration test, system test, or approval test as shown in Figure 2.1. Thus, the tester considers which piece of software to test which can be just a single function (method), a complete class, a collection of functions or classes (library), or a whole application with its internal and graphical interfaces. For instance, *integration testing* checks the communication between different modules to ensure that they interwork correctly.

System Knowledge

The amount of knowledge about the underlying system determines which kind of test and test architecture can be used. There are three distinguishable testing types:

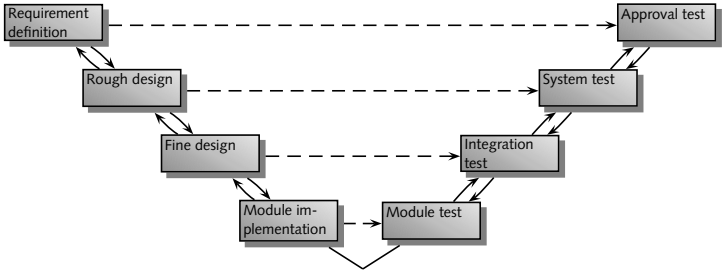


Figure 2.1.: *The V process model for software development (Balzert 1998, page 101)*

Black-box testing is applied to systems about which no internal knowledge is available. Only the interfaces to the environment are accessible. In addition, the specification is available from which tests can be generated. Hence, test data can be given as input and the output data can be evaluated against the specification. Black-box testing is mainly done in the later development stages.

Grey-box testing is used if some internal knowledge is available. The knowledge is used to improve code coverage for black-box testing.

White-box testing, also called glass-box or structural testing, is used if full knowledge about the implementation is available. For instance, the source code is available. Hence, the information can be used to center testing, to control code coverage, to check value boundaries, and to do algorithmic specific testing. Furthermore, knowledge about control flow and data integrity can be used to design tests. White-box testing is mainly done in the implementation phase.

Test Objectives

The application area influences the requirements which have to be fulfilled. Thus, there are also different kinds of tests.

Functional testing is used to check the behaviour with regard to the functional requirements. Test data are sent to the system and the received output is checked against the specification. The term functional testing is sometimes used as synonym for black-box testing.

Performance testing is used to check non-functional requirements like response time or memory usage in normal or overload situations. There are three special cases:

Real-time testing checks some real-time requirements. They can be divided into hard and soft requirements. Hard requirements concern definite time boundaries which have to be fulfilled. Soft requirements describe time boundaries or some variances which may be violated within a definite range.

Load testing checks the behaviour when heavy load is generated. For instance, the response time is checked.

Stress testing checks the behaviour under unusual conditions by, for instance, sending inopportune or malformed data.

Portability testing is used to check the portability to other environments like other hardware or software platforms. For example, operation system derivates have to be checked against the different environments.

Penetration testing is used to check the vulnerability of a system like whether it is possible to get unauthorised rights, wrong configurations can lead to successful attacks, or are there any known bugs in used software.

Usability testing checks (graphical) user interfaces with regard to criteria such as adequacy, simplicity, clarity, and consistency. Typically, usability tests are performed by recording and evaluating the behaviour of an external test person interacting with the system (gestures, response times, eye movement, etc.).

Test Data Selection

Testing is always concerned with feeding a system with input and evaluating the output. Hence, the method to choose the test data is important.

Exhaustive testing means to enter all possible data on each possible input request. This would be a complete test regarding the claimed input but it is only applicable for small amounts of input requests or possible test data because of the increasing complexity for a bigger amount.

Partition testing segments the input domains into sub-domains without overlapping. For instance, boundary testing to check boundaries and extreme values.

Random testing refers to the random choice of test data from the whole set of data.

Mutation testing is based on the approach that many different system versions, called mutations, are generated. Each mutation is the result of a small modification like removing or modifying statements. Test data have to be selected in such a way that all modifications are detected. Hence, the probability could increase to get good test data for the original system.

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 checks the underlying system against its specification. This implies black-box testing.

Interoperability testing or compatibility testing checks the interworking of different implementations of the same specification. For instance, protocol implementations from different companies. Conformance to the specification does not ensure interoperability because the specification can be, for instance, incomplete and hence interpretable. Likewise, interoperability does not ensure conformance to the specification, because both implementations could misinterpret the specification in the same way.

There are two kinds of tests, namely *active* and *passive* interworking tests. Passive tests check only valid behaviour whereas active tests allow to introduce errors, wrong behaviour, wrong data, etc.

Regression testing is used if a part of a system has been modified and the behaviour of the whole system has to be tested. To detect side effects of the modifications, only outputs have to be considered which differ in comparison to output from the old system.

Comparison testing compares the output of different systems which are based on the same specification. The aim is to find output differences.

2.2. Problems of Object-Orientation for Testing

The main idea of object-orientation is to bind data and belonging methods together and *encapsulate* them in classes. Additionally, to provide more expressive mechanisms like *inheritance* and *polymorphism* to enhance reusability. It was believed that the new paradigm also leads to less faulty systems and is easier for system testing because of more compact code. However, object-oriented software is still produced in the same imperfect way as before. For instance, with the same humans and similar developing methods. Despite of the expectations the object-oriented paradigm has its own kinds of pitfalls because of powerful concepts like inheritance, polymorphism, late binding, and encapsulation. Consequently, the advantages of object-orientation with regard to development are disadvantages for testing. Therefore, testing object-oriented systems is still necessary and effective testing requires special attention to object-oriented pitfalls (Binder 2000; Kung et al. 1998; McGregor & Sykes 2001).

The test model and consequently the test strategy depends on the test aim (ISO/IEC 1994). For instance, a system crash must not be a reason to fail a functional test but it would be a reason for a fault directed test. Nevertheless, knowledge about object-oriented faults is useful for both cases. The paradigms like encapsulation, object composition, and complex runtime behaviour by polymorphism are an obstacle for testing. Test case design and coverage analysis are difficult wherefore automatic generation tools have to provide special support for object-orientation. Interaction is described by a complex set of message sequences and states. Polymorphism and dynamic binding increase number of execution paths. Furthermore, objects and consequently object states are distributed over the whole system which makes state control difficult. Inheritance has the effect that correctness of a superclass does not lead to correctness of a subclass of it. Additionally, reusability of a class requires careful testing and retesting in each new context. Interfaces are used quite a lot wherewith more interface faults occur.

An error and a failure list and a method scope fault taxonomy is given in Binder (2000, section 4.2.7). Some language-specific hazards for C++, SMALLTALK, and JAVA are also given in Binder (2000, section 4.3). Some details of the above mentioned problems are detailed now.

Encapsulation

Information hiding and modularity are achieved by encapsulation of data and methods in classes where access is controlled. Therefore, dependencies and global access are prevented by hiding implementation. Encapsulation is not directly related to faults but it is a obstacle for testing because abstract and concrete states have to be influenced by testing. However, direct access to states by, for instance, *get* and *set* methods is often not possible.

Inheritance

An essential aim of object-orientation is to support reusability by using elements of a common entity. For instance, types and subtypes with extensibility can be defined. If inheritance can be mirrored in the test suite, the test effort for a subclass can be reduced. Inheritance can be a very powerful means but some weaknesses and misuses can lead to a lot of trouble:

- deep (hierarchy of subclasses) and wide (usage in many classes) inheritance,
- inheritance weakens encapsulation because subclasses can get direct access to superclass elements (hence, contract of superclass could be violated by subclass),
- participation in implicit control mechanism for dynamic binding because of unanticipated bindings or misinterpretation of correct usage,
- abuse by using as macro expansion mechanism, and
- as a model of hierarchy where no sharing or using is done.

Thus, faults by side effects, inconsistencies, and incorrect behaviour have to be considered.

Incorrect initialisation of objects by missing initialisation in the superclass, modified super initialisation, or forgotten overwrite of methods like **copy** and **isequal** lead to faults. Especially in retesting modified methods/algorithms, the dependencies between classes have to be considered. Mixing problem domain relationships and shared implementation features are problematic. For instance, class/subclass and type/subtype relationships require careful handling. A subtype has a specification relationship and a subclass has an implementation relationship. Subclasses have to be tested

against their specifications and against the specification of the superclass. Reuse is difficult if it was not intended initially because of assumptions or optimisations specific to the original application. Identically named methods from different classes can produce faults because of incorrect dynamic binding or when used from another class. Scoping rules influence binding wherefore usage in subclasses may fail.

Abstract and generic classes require the creation of a concrete class to get tested. For generic classes, the interaction between class and used type has to be tested. However, an exhaustive testing of all types for a generic class is not feasible and would be equal with formal verification.

Polymorphism

Binding a reference to more than one possible object or method allows for compact, elegant, and extensible code and is called polymorphism. Binding and type checking at compile time is called static polymorphism. At runtime, it is called dynamic polymorphism respectively dynamic binding.

Semantics, syntax, and binding search mechanisms to select a method at runtime differ between programming languages. Polymorphism makes code difficult to understand and error-prone because behaviour is not predictable by a static analysis and the code is hard to read. Thus, wrong method binding has to be excluded by testing. Many variables, which influence polymorphic methods, are not visible in source so that it is difficult to understand all possible interactions with all bindings. Dependencies among polymorphic methods are stronger than for normal methods because of their wider application. Therefore, method specification modifications have more influence and unmodified classes lead to faults. Method redeclaration in subclasses is dangerous especially in the context of polymorphism.

Consequently, usage of polymorphism can be fault-prone. Common mistakes are ignoring responsibility, independent revision of its definition and usage, contract inconsistency, not provided method, method misuse, or incorrect interface signature.

Interaction

Classes are collections of methods and states and they interact by calling methods where interpretation depends on the current state. However, which are correct sequences of method calls? A corrupt state can be occur by faulty method interworking or method implementation. Method

interworking may be faulty by overlapping responsibilities or if there is a concrete sequence pattern to produce a corrupt state. For instance, overlapping responsibility is given if an internal variable is modified by several methods without considering the implications. Method implementation may be faulty by using a corrupt algorithm or giving an incorrect output. Furthermore, a method implementation can be overridden, for instance, by bad inheritance, or a wrong contract is implemented.

In case where sequences lead to the same result, an equivalent sequence set is found. If different instances do not produce the same result, a fault is found. State rules of objects have to be considered and thus, illegal method calls in definite states are forbidden. This is necessary to prevent wasting computation time and to provide a stable system. However, a defensive system design should consider wrong method calls and forbidden method calls have to be tested. Methods have to be tested in cooperation because testing a method alone is not enough. A fault could occur later during using another message.

Services

There are services provided by the programming language and used compiler respectively which have also to be considered for testing. Default services like providing default constructor, deconstructor, or copy constructor are supported. Runtime conversion services for classes, which are similar to type conversion, are used to convert superclass objects to subclass objects and vice versa. Garbage collection services could cause problems under high loads. Providing an object identity during run-time to distinguish types, subtypes, and objects is only done for subclasses by the programmer itself (via **copy** and **isequal** methods).

Summary

To sum up, static testing is less effective for object-oriented systems because of dynamic effects like late binding and coding complexity like inheritance and polymorphism. An effective test process for object-oriented systems has to consider these problems (Binder 2000; Kung et al. 1998; McGregor & Sykes 2001) and thus,

- design for testability in all phases is important,
- testing must adapt to iterative and incremental development,

- test design has to consider methods, classes, and clusters at the same time because testing a cluster of classes and not a class alone is necessary to consider the environment of a class,
- test suite structure should correspond to SUT structure which enhances readability, maintenance, and automatic generation.

Thus, black-box testing like component testing is not enough because specific bugs of object-oriented systems are missed and methods using code coverage analysis have to consider object-orientation, too.

2.3. Test Generation

As discussed before, testing has to be integrated into the development process. Therefore, to reduce testing effort tests (test purposes and test cases) should be generated from system specification which is named CATG.

Manual test generation is error-prone wherefore test generation must be automated to be effective and repeatable. Efficiency allows a quick validation which speeds up debugging. Repeatability, through usage of test cases once more, enables more often testing wherefore minor system modifications can be validated immediately. Furthermore, automation leads to consistent test results which eases test result analysis. The uniform test process is independent of the responsible person for testing. Using automation enables better productivity because test staff can concentrate on test design and not test execution and test suite maintenance. The selection of test method and tools for automation depends on test experience, test goals, budget, used software process, kind of application under development, particulars of the development and target environment, etc.

Automated testing can permit test execution of long and complex tests, automate comparison for many test outputs to evaluate test results, and automatic adaptation to different versions of the SUT. For instance, regression testing benefits a lot from automation. Automatic testing involves running test suites without manual intervention and generation of test inputs and expected results.

However, manual test generation is also useful if, for instance, many user interaction is necessary, no repeat is necessary, automatic generation is too expensive, or full automatic generation is not possible or not effective. Skilled testers with good knowledge of SUT can develop good but limited test cases where focus is mostly on specific scenarios. Thereby, combining manual and automated testing is quite common. Thus, it is an

advantage if manual tests are written in the same language or languages as the system specification. It provides seamless integration between manual and automatic test generation and improves maintenance and readability.

In the section remainder some remarks to the tools AUTOLINK and TESTCOMPOSER are given which provide manual and automatic test generation facilities (Schmitt et al. 2000). Firstly, an introduction is given. Secondly, an overview about the test generation process is given. Lastly, scenario-based testing by direct translation of MSCs is described. Test purpose based testing and test case generation with AUTOLINK are described in Koch (2001) and automatic test generation using state space exploration with AUTOLINK based on formal specifications is described in Schmitt (2003).

Autolink and TestComposer

In many cases, a formal specification of the SUT is given in the *Specification and Description Language* (SDL) (Ellsberger et al. 1997; ITU-T 1999). SDL not only allows to describe the structure and behaviour of a communicating system in a semi-graphical way; there also exist tools for dynamic analysis of SDL specifications by means of simulation and validation. Hence, a reasonable approach is to generate test cases automatically based on a given SDL specification. In addition to an increased efficiency in terms of both time and cost, automatic test generation ensures consistency between the formal specification and the test cases applied to an implementation.

For that reason, the two major SDL tool vendors TELELOGIC and former VERILOG have integrated automatic test generation tools into their software development environments.[2] TELELOGIC complemented its TAU tool suite with AUTOLINK in 1997. AUTOLINK has been developed at the Institute for Telematics, Medical University of Lübeck (Koch 2001; Schmitt 2003) and is based on the former work of the SAMSTAG project (Grabowski et al. 1997). In 1998, VERILOG extended OBJECTGEODE with TESTCOMPOSER. Similar to AUTOLINK, it has its root in the research area as it is based on TGV and TVEDA which were developed at IRISA/VERIMAG and FRANCE TELECOM/CNET (Kerbrat et al. 1999).

Both tools share the same basic concepts. For example, they apply state space exploration techniques to search for suitable test sequences. In addition, they support the second edition of the standardised TTCN (ISO/IEC

[2] In December 1999, the two companies have merged.

1998b) as a common output language. Nevertheless, many concepts are realised differently in TestComposer and Autolink. Moreover, the two tools put their focus onto different steps of the test generation process. The strengths of TestComposer are in the flexible specification of test purposes whereas Autolink has its strong points when it comes to the customisation of the generated TTCN test suites.

TestComposer and Autolink have been described separately in detail in former publications (Grabowski et al. 1999; Kerbrat et al. 1999; Koch et al. 1998). A short introduction to the overall process of test generation is given in order to make the reader familiar with the general approach.

Overview

Autolink and TestComposer are tightly integrated into their corresponding development environments. TestComposer is built on top of the ObjectGeode Simulator; Autolink is part of the TAU Validator. In this way, the tools can make use of the functionalities of their underlying applications. The Simulator as well as the Validator are used to find dynamic errors and inconsistencies in SDL specifications. They provide roughly the same basic features with state space exploration as their fundamental concept.

Test generation with TestComposer and Autolink follows a three-stage process. An overview is given in Figure 2.2. In the diagram, actions are represented by rounded boxes. Data structures and files are depicted in rectangles. Finally, configuration scripts that influence the test generation are indicated by hexagons.

In a first step, the user has to specify a set of test purposes. Each test purpose defines a specific aspect of the behaviour of the implementation that is intended to be tested. With regard to TestComposer and Autolink, a test purpose is considered to be a sequence of input and output events that are to be exchanged between the given SDL system and its environment. Test purposes are developed either manually by using, e.g., an MSC editor, interactively by stepwise simulation of the SDL system, or fully automatically.

There are different representations for test purposes: Autolink uses *Message Sequence Chart-1996* (MSC-96) (ITU-T 1996) as an uniform format. TestComposer uses MSC-96 as well but also creates scripts in a proprietary format that can be handled by ObjectGeode more efficiently than MSCs. Both tools support observer processes which are similar to

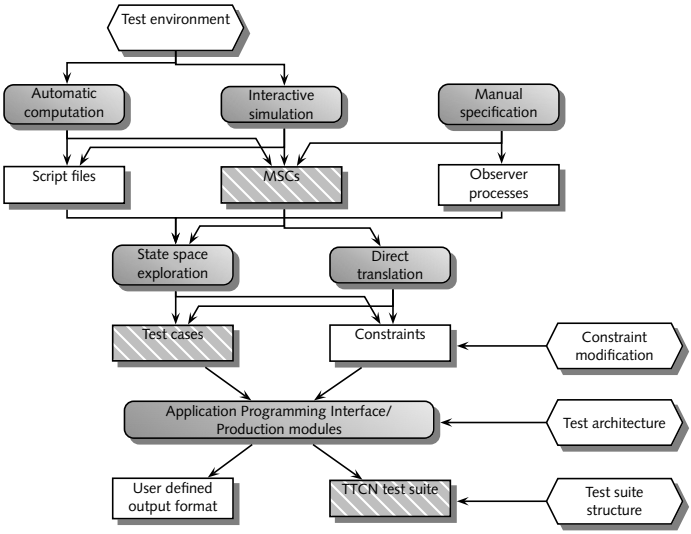


Figure 2.2.: *Test generation with* TestComposer *and* Autolink

regular SDL processes. They run in parallel with the actual SDL system and allow to inspect and control its simulation.

Based on a set of test purposes, test case generation takes place. Normally, a generation engine computes a test case based on state space exploration of the SDL system. By this, it can determine additional valid interactions between the tester and the SUT which are not already specified in the test purpose description. However, sometimes it is not possible to simulate a test purpose. For these cases, Autolink provides a way to translate test purposes directly into test cases.

All test case descriptions along with their *constraints*, i.e. the definitions of the data values exchanged between the tester and the SUT, are stored in an internal data structure. Autolink allows to save and reload generated test cases to disk such that the user can suspend and continue the generation of a full test suite.

In a final step, Autolink produces a test suite in TTCN-2 format. TestComposer provides an *Application Programming Interface* (API) (which is public) that allows customers to adapt the tool to any arbitrary test specification language. In addition to the internal test case repres-

entations, the API provides access to general information about timers, signal types, etc. and *Point of Control and Observation* (PCO). TestComposer already includes a module that produces test suites for second edition TTCN. In the following, only TTCN will be considered for output as many features of the tools are related closely to this notation.

Test generation with Autolink and TestComposer is influenced by a number of configuration settings. For example, when generating test purposes (semi-) automatically, the developer has to provide the simulator with information on the test environment of the system, i.e. reasonable input values. The look of the test suite can also be controlled by various options. In Autolink, constraints can be named and parameterised by user-defined rules. In addition, test cases can be combined in a hierarchy of test groups to express their relationships. Last but not least, the test architecture has a great impact on the final test descriptions. A test case that is executed on a monolithic tester will look differently from a test case that is designed for a distributed test system.

Scenario-based Testing

If a test purpose defined as MSC covers certain aspects of a protocol specification which are not represented in the corresponding SDL model or if a SDL model is missing completely, it is obviously not possible to generate a test case by state space exploration. To handle these cases, Autolink provides direct translation of MSCs into TTCN test cases with consistency checks regarding the SDL system interface definitions. Hence, an SDL system has to be provided which at least defines the channels to the system environment in order to identify the PCO and the signals sent via these channels.

Direct translation of MSCs into TTCN test cases has to be applied with caution. There is no guarantee that the MSCs and hence the test cases describe valid traces of the specification or the implementation, respectively. Instead, Autolink relies on the developer that the test cases are valid. Furthermore, it is not possible to compute test events which lead to an *inconclusive* test result, meaning any deviation from the behaviour described in the MSC is considered to be false.

On the other hand, there are good reasons to specify MSC test purposes instead of directly writing TTCN test cases. Firstly, test cases typically span trees with several tree leaves because of the partial order of test events. In MSCs, the partial order is expressed inherently due to the semantics of MSC. While it is arduous for a test suite developer to write

down a complete TTCN test case, AUTOLINK automatically computes all valid permutations of test events for a given MSC.

Secondly, since AUTOLINK always translates MSCs into an intermediate internal test case representation, test cases generated by an MSC to TTCN translation can be merged with test cases generated by state space exploration. This leads to uniform and compact test suites with a reduced number of constraints.

2.4. Testing and Test Control Notation

The *Tree and Tabular Combined Notation* (TTCN) is the third part of the *Conformance Testing Methodology and Framework* (CTMF) (ISO/IEC 1994) standard for the specification of test suites for conformance testing. In May 2001, the new version of TTCN, called *Testing and Test Control Notation (version 3)* (TTCN-3), was finally standardised (ETSI 2002a). The TTCN-3 (ETSI 2002a) is a universal and standardised language for the specification and implementation of tests for distributed systems.

TTCN-3 is the target language for test generation within the scope of this thesis. TTCN is widely accepted in the area of testing telecommunication protocols. Contrary to existing programming or scripting languages like C and the DEJAGNU GNU Testing Framework (Savoye 2001) or test frameworks like XUNIT, the TTCN-3 provides an appropriate level of abstraction, high-level testing concepts, and control structures. Hence, writing abstract and implementation independent test suites gets possible which widens the application area. It is also easier to read and write tests and to provide standardised test suites for standardised protocols. Furthermore, test engineers have to learn only one test language and can mostly use the same testing tool set. TTCN-3 tools are offered, for instance, by TELELOGIC, TESTING TECHNOLOGIES, and DA VINCI COMMUNICATIONS which support editing, compilation, debugging, and execution of TTCN-3 modules.

Improvements

TTCN-3 is called the successor of TTCN-2 (ISO/IEC 1998b) but it was redesigned from scratch and uses another style. TTCN-3 improves concepts of TTCN-2 and introduces new concepts to support a broad spectrum of testing types, e.g., conformance and interoperability testing, and its communication mechanisms allow for testing various platforms such as the CORBA or Internet-based protocols. An important feature of TTCN-3

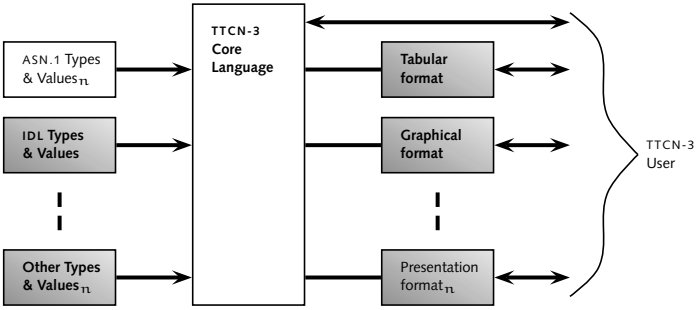


Figure 2.3.: *User's view of* TTCN-3 *core language, presentation formats, and imported types*

is the enhanced communication concept which now supports procedure-based communication to provide synchronous communication, as well as the asynchronous message-based communication. Additionally, a test execution control part, a module and grouping concept, and new data types are introduced to provide better control and grouping mechanisms.

TTCN-2 was designed to test networks which are conform to the *International Organisation for Standardisation* (ISO) *Open Systems Interconnection* (OSI) reference model. The OSI terminology and concepts like *Abstract Service Primitive* (ASP) and *Packet Data Unit* (PDU) and conformance testing peculiarities have been removed as far as required to widen applicability of TTCN-3. Additionally, *constraint* handling was replaced by *templates* which provide parameterisation and matching mechanisms. Use of data types defined via ASN.1 is also possible in TTCN-2. However, TTCN-3 has integrated some ASN.1 data types into the language itself and allows import of ASN.1 data types.

As the name TTCN states, a tabular form was used in TTCN-2. However, TTCN-3 abandons the tabular form and uses instead a text-based language which is comparable to an implementation language like C. This new core language is used as base for document interchange and also for a tabular presentation format defined in ETSI (2001). A graphical presentation format is also defined in ETSI (2002b) (see Figure 2.3) which is called TTCN-3 *Graphical Presentation Format* (GFT).

The remaining part of this section shall describe some basic concepts of TTCN-3 itself to set a base for the following chapters. This includes the module and group concept, the data concept, communication, test

configuration, templates, and behaviour description. The *Inres* example gets described first.

2.4.1. Inres Case Study

The concepts of TTCN-3 are illustrated by test suites for *Inres*, a service and protocol designed for educational purposes (Hogrefe 1989). It is also used in section 3.4 for the description of the formal specification language MSC.

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 MSC in Figure 2.4.

The main features of 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 behaviour 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 2.5.

In the following sections, only simplified extracts are presented. A complete test suite can be found in Appendix B (Schmitt 2003).

2.4.2. Structuring

Modules are the top-level structuring element in TTCN-3 and a TTCN-3 document is composed of one or more modules. Each module represents either a complete executable test suite or a library. It consists of *definitions* and an optional *control* part that guides the execution of test cases. Modules support usage of parameters to permit the re-use of modules in different test environments. A module can *import* definitions from other

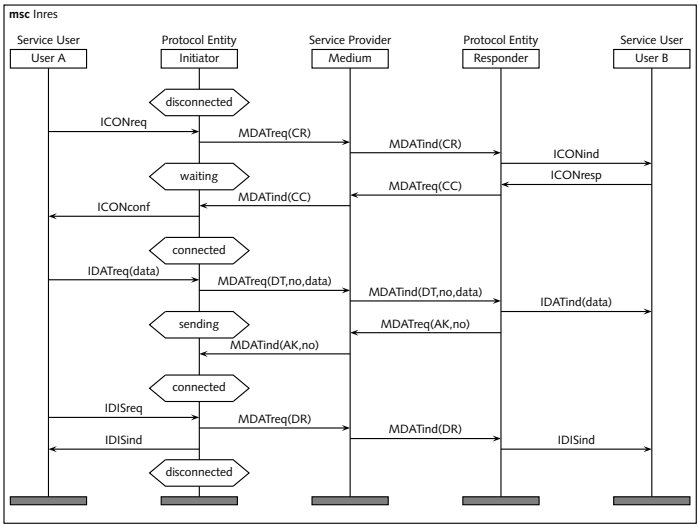


Figure 2.4.: *The Inres service and protocol*

modules but it cannot import their control parts. Unfortunately, modules cannot be nested.

Definitions in the definitions part include constants, data types, communication data such as messages, signatures, and templates, test configuration elements like ports and components, and dynamic behaviour by definition of test cases, altsteps, and functions. The declaration of variables in the definition part is not supported whereby no global variables, timers, etc. are available.

Definitions can be combined into groups, but a group does not define a new scope and has no semantics purpose except when definitions are imported by another module. Groups are used to structure test data in a logical manner and to enhance readability.

In the module control part test case execution and their execution order is given why it can be seen as the *main* method respectively program of the module. Test case execution order can be controlled by dependencies from results of other test cases or timers, for example.

In Listing 2.1, a TTCN-3 module for testing conformance of an Inres Initiator protocol entity is presented. Its definition part starts with an

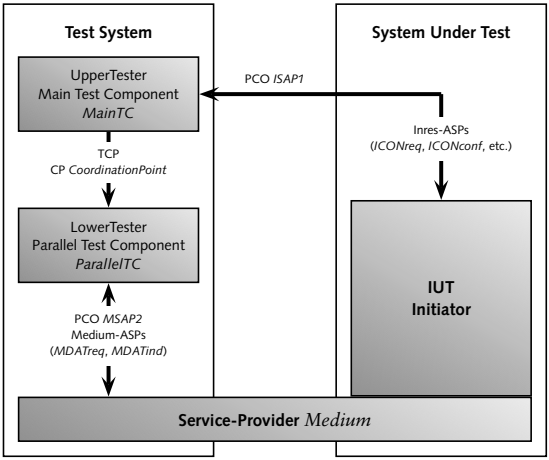


Figure 2.5.: *The local test method applied to Inres*

import statement (line 2–5) to adopt data type *UserPDU* and constant *someUserPDU* from an external module called *ServiceUser*. All data type definitions that are required to describe an Inres PDU are combined in group *BasicDefinitions* (lines 7–17). Thereafter, global constant *maxTestCaseTime* is declared in line 19. Module *TestsForInres* includes many more definitions. For better readability and comprehension, these definitions are presented separately in Listings 2.2–2.8.

In the module control part (lines 23–29), 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.

2.4.3. Data Concepts

TTCN-3 provides its own data type model which was inspired by ASN.1 and programming languages. The types are listed in Table 2.1. Most basic types such as **integer**, **char** (see ISO/IEC 1990), **universal char** (see ISO/IEC 1993), and **boolean** are well known from programming languages. The basic string types differ only in the used character set where **charstring** and **universal charstring** are based on the same character sets as **char** and

Listing 2.1: TTCN-3 *Module* TestsForInres

```
module TestsForInres( integer maxRepetitions, boolean testInopportuneEvents ) {
  import from ServiceUser language "ASN.1:1997" {
    type UserPDU;
    const someUserPDU;
  }

  group BasicDefinitions {
    type UserPDU InresSDU;
    type enumerated InresPDUType { CR(1), CC(2), DR(3), DT(4), AK(5) };
    type enumerated SequenceNumber { zero(0), one(1) };
    type record InresPDU {
      InresPDUType iPDUType,
      SequenceNumber seqNo optional,
      InresSDU iSDU optional
    }
    type InresPDU MediumSDU;
  } with { encode "PER—BASIC—UNALIGNED:1997" } // apply Packed Encoding Rules

  const float maxTestCaseTime := 50;

... further definitions ...

  control {
    var verdicttype overallVerdict := pass;
    overallVerdict := execute( SingleDataTransfer (), maxTestCaseTime );
    if ( overallVerdict == pass and testInopportuneEvents == true ) {
      overallVerdict := execute( DataLoss() );
    }
  }
} with { encode "BER:1997" } // apply Basic Encoding Rules by default
```

universal char. The TTCN-3 special basic type **verdicttype** is used to handle test verdicts where only the five distinguished values **pass**, **fail**, **inconc**, **none**, and **error** are available for it. Type **objid** is used as object identifier and is imported from ASN.1.

Available structured base types are **enumerated**, **record**, **set**, **union**, and **array**. The type **record** is an ordered type whereas **set** is an unordered type which is important in case of data encoding. Apart from order, both are equal and provide *optional* fields. In case of using only a single type, the types **record of** and **set of** are available. They can be considered similar to an ordered and unordered array respectively.

In case of handling data of unknown type the data can be assigned to type **anytype** which is shorthand for the **union** of all *known types* in a TTCN-3 module. The **anytype** was especially introduced to provide a better mapping of IDL (see chapter 5) which was proposed by the author.

TTCN-3 provides three special configuration types where type **address**

Table 2.1.: *Overview of* TTCN-3 *types*

Class of Type	*Type (Keyword)*
Simple basic	integer
	char
	universal char
	float
	boolean
	objid
	verdicttype
Basic string	bitstring
	hexstring
	octetstring
	charstring
	universal charstring

Class of Type	*Type (Keyword)*
Structured	record
	record of
	set
	set of
	enumerated
	union
Special data	anytype
Special configuration	address
	port
	component
Special default	default

is used to address entities inside the SUT. Test configuration is organised via type **component**, and type **port** is used to handle communication between components which is described in more detail in subsection 2.4.5 on page 31.

Lastly, the type **default** is mentioned which is used to handle default behaviour defined by altsteps which is described in detail on page 35.

The test specifier may define own types by sub-typing of types. The set of valid values of basic and structured types can be restricted by usage of value ranges, lists of values, and length restrictions. The data model defines no dynamic types why no pointers are available. However, recursive data structures can be used, instead.

A set of predefined functions is available to support data value conversion like integer to string, to get the number respectively length of records, sets, and strings, to check the presence of optional fields, to check the chosen type in unions, to retrieve substrings, and to generate random numbers (ETSI 2002a, appendix C).

Attributes

TTCN-3 provides assigning of attributes to statements to provide, for instance, additional information for compilers or other tools like graphical editors or viewers. The provided attributes are **display**, **encode**, **variant**, and **extension**. Attribute **display** is used for presentation purposes and attrib-

Table 2.2.: *Overview of* TTCN-3 *type variants respectively* useful *types*

Base Type	*Variant*	*Useful Type*
integer	8 bit	byte
	unsigned 8 bit	unsignedbyte
	16 bit	short
	unsigned 16 bit	unsignedshort
	32 bit	long
	unsigned 32 bit	unsignedlong
	64 bit	longlong
	unsigned 64 bit	unsignedlonglong
float	IEEE754 float	IEEE754float
	IEEE754 double	IEEE754double
	IEEE754 extended float	IEEE754extfloat
	IEEE754 extended double	IEEE754extdouble
universal	UTF-8	utf8string
charstring	UTF-16	utf16string
	UCS-2	bmpstring
	8 bit	iso8859string
record	IDL:fixed FORMAL/01-12-01 v.2.6	IDLfixed

ute **extension** is used for user-defined extensions. The encoding attributes **encode** and **variant** define encoding rules and encoding variants respectively.

Especially the encoding attributes are important for data types because encoding is important for data transmission and variants are important to specify well defined sub-types. TTCN-3 itself specifies no such implementation specific information.

There is a set of predefined variant attributes available. The usage of these variants to define *useful types* is shown in ETSI (2002a, appendix E) and is listed in Table 2.2. They were proposed by the author to improve mapping of IDL (see chapter 5).

Type Import

Sometimes it is useful to import existing type and data definitions from other sources like the SUT implementation or specification. Therefore, TTCN-3 provides the possibility of *importing* definitions defined in another language than TTCN-3. Until now, only import rules for ASN.1 are supported (ETSI 2002a, appendix D). ASN.1 is heavily used in telecom-

munication applications it was also supported in TTCN-2. There exist many SDL specifications for telecommunication applications and therefore, usage of SDL is also interesting. However, until now there are no import rules defined.

Nevertheless, for system specifications using IDL the author has defined explicit mapping rules which can be seen in chapter 5 and in ETSI (2003).

2.4.4. Communication

TTCN-3 distinguishes between message-based and procedure-based communication which could also be called asynchronous and synchronous communication respectively. Communication is used between test system and SUT and within the test system itself.

Message-based communication is done by **send** and **receive** operations and is based on asynchronous message exchange where only the receiver gets blocked (see Figure 2.6). The transferred data can be defined by any type but typically records are used.

Procedure-based communication is used to call procedures in remote entities like it is done in *Remote Procedure Call* (RPC), CORBA, and *Distributed Common Object Model* (DCOM). On caller side the communication is handled by operations **call**, **getreply**, and **catch** and on callee side by operations **getcall**, **reply**, and **raise**. Procedure calls in general may block on the calling and called side (see Figure 2.6). In TTCN-3 the called side gets always blocked and blocking on the caller side is adjustable. Non-blocking procedure calls marked by the keyword **noblock** have some limitations because no values can be transmitted from the called side in context of the procedure call. Furthermore, blocking of procedure calls marked by keyword **nowait** may be ignored any time whereas continuing is possible at all times. In contrast to **noblock** procedure calls, **nowait** procedure calls have no limitations because a possible response may be handled afterwards.

Signatures

The information to be transmitted or to be received in sending or receiving operations for procedure-calls are defined by (inline) *signatures*. Using signatures enables semantics checking of corresponding communication operations. Signatures consist of a parameter list, return value, exception list, and blocking characteristic (default is blocking), as demonstrated below. The **signature** parameter list includes identifier, type, and direction as

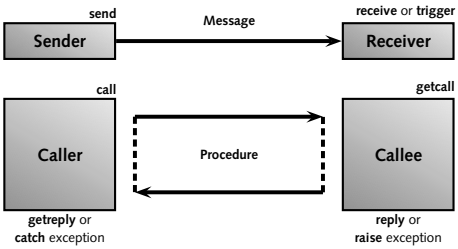


Figure 2.6.: *Message- and blocking procedure-based communication*

used in IDL (see subsection 3.5.5). Parameters with direction type **in** have call-by-value semantics and parameters with direction types **inout** and **out** have call-by-reference semantics.

TTCN-3

```
signature MyBlockingProcedure (in integer par1, inout float par2, out float par3)
          return integer exception (Excep1, Excep2);
signature MyNonblockingProcedure (in integer par1) noblock;
```

Templates

Templates handle distinct values to be sent or received. They are used to organise and re-use test data by providing a structure to define them. Templates can also be used inline to enhance readability and to avoid unnecessary expense in case of empty templates or templates with only few fields. *Signature templates* are used for procedure-based communication and *type templates* are used for message-based communication.

Templates provide parameterising, referencing by using other templates, and modification by extending templates. Values, value ranges, and matching mechanisms can be used in templates. At time of sending, templates have to define concrete values why ranges and matching expressions have to be fully resolved. If used in receive operations, received data is tested against the used template where templates with value ranges and matching mechanism simplify testing. Hence, templates are very powerful, but the matching expressions could be improved as described in Schmitt & Ebner (2003). *Type templates* are mostly used together with records where all required values are defined.

Listing 2.2: TTCN-3 *Templates*

```
1  template MDATind ConnectionRequest := {
2    mSDU := { iPDUType := CR, seqNo := omit, iSDU := omit }
3  }
4
5  template MediumSDU ConnectionConfirmation := { // this template is used with
6    iPDUType := CC, seqNo := omit, iSDU := omit    // template 'MediumDataRequest'
7  }
8
9  template MDATreq MediumDataRequest( template MediumSDU data ) := {
10   mSDU := data
11 }
12
13 template MDATind DataTransfer( InresSDU data ) := {
14   mSDU := { iPDUType := DT, seqNo := ?, iSDU := data }
15 }
```

In Listing 2.2, various templates are defined that are used in function *MediumAccess* (see Listing 2.7). 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 Listing 2.7, line 6.

The two templates *ConnectionConfirmation* and *MediumDataRequest* (Listing 2.2, lines 5–11) illustrate the dynamic chaining of templates. *MediumDataRequest* is parameterised by a template of type *InresPDU*. In Listing 2.7, line 8, it is instantiated with template *ConnectionConfirmation* as actual parameter. Template *DataTransfer* (lines 13–15) 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* (Listing 2.6, lines 17, 18, 25, 26, 32, and 33) and function *MediumAccess* (Listing 2.7, lines 13, 25, 26, and 28). 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.

Ports

Communication under TTCN-3 is characterised by communication endpoints which are called *ports*. Each communication operation is directed to a port and not to a connection. For each port there is a list defined where all allowed message types and procedures including their direction

are given. A port allowing message- and procedure-based communication is of port type **mixed** and of type **message** or **procedure** respectively if only one communication type is allowed. Each message type or procedure has one of the three directions **in**, **inout**, or **out** defined. Hence, fine control of communication directions is possible which was not defined in TTCN-2.

Ports are modelled as an infinite FIFO queue where all incoming messages or procedure calls are stored until explicitly consumed or removed by the owner of the port. Since there is no automatic removing, each incoming calls have to be handled. Hence, specified tests have to consider all incoming calls and if a call is not necessary for the test result it has to be explicitly ignored by removing.

There are several operations available to control ports. The top element of a port queue can be checked without removing it. Port queues can be cleared, started, and stopped during execution of a test. The receiving operations **receive**, **getcall**, **getreply**, and **catch** consume the top corresponding element of the port queue if matching was successful. Despite the **receive** operation the **trigger** operation consumes each incoming message even in case of a mismatch. In case of a match it works like a **receive** operation. Hence, searching or waiting for a specific message gets easily possible.

In Listing 2.3, 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 4–7. 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 9–15. A concrete message exchange is described in the test case shown in Listing 2.6. In lines 17, 18, 22, 32, and 33 various messages are sent and received at port *ISAP1* which is of type *InitiatorSAP*.

A simple example of procedure-based communication is introduced in Listing 2.4. 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. [3] Two contrary port types are defined for it: *PortAtMTC* (lines 3–5) accepts incoming calls, whereas *PortAtPTC* (lines 7–9) is used to invoke the remote procedure. A concrete procedure call is realised in Listing 2.6, lines 25 and 26, and Listing 2.7, lines 25 and 26.

[3] 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.

Listing 2.3: TTCN-3 *Message-based Communication*

```
type record ICONreq {};
type record IDATreq { InresSDU iSDU };

type port InitiatorSAP message {
  out ICONreq, IDATreq, IDISreq;
  in ICONconf, IDISind;
}

type record MDATreq { MediumSDU mSDU };
type record MDATind { MediumSDU mSDU };

type port MediumSAP message {
  in MDATind;
  out MDATreq;
}
```

Listing 2.4: TTCN-3 *Procedure-based Communication*

```
signature acknowledgementSent();

type port PortAtMTC procedure {
  in acknowledgementSent;
}

type port PortAtPTC procedure {
  out acknowledgementSent;
}
```

2.4.5. Test Configuration

TTCN was designed for functional, black-box testing and to describe *Abstract Test Suites* (ATSs) which are independent of concrete test platforms. Therefore, special interfaces between SUT and ATS are required to make a test suite executable. Furthermore, a distributed and concurrent test architecture is used and TTCN-3 additionally supports *dynamic* test configuration, whereby test configuration can be modified during test execution.

Test configuration in TTCN-3 is structured by test *components* which are interconnected by well-defined communication endpoints called *ports* (in TTCN-2 via *Communication Points* (CPs)). There is exactly one *Main Test Component* (MTC) which controls all other test components called *Parallel Test Components* (PTCs). PTCs can be dynamically created whereas the MTC is created automatically at each test case execution. The PTCs are independent of each other. Termination of the MTC automatically terminates all PTCs. A test component terminates by leaving its **function** or **testcase** respectively, executing a **stop** command, or getting terminated by another test component. Test components can have local timers, variables, and constants. Ports are allowed to have one-to-many connections to support multicast connections.

The environment of the test system is defined by an *Abstract Test System Interface* (ATSI) which specifies all communication endpoints to the SUT in an implementation independent manner. Hence, communication

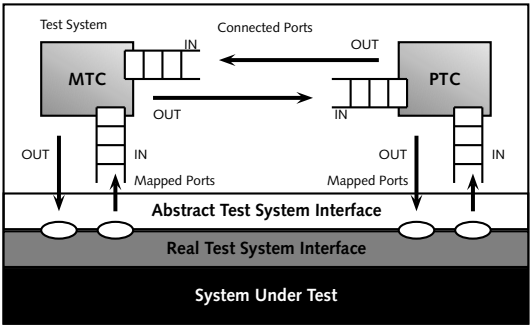


Figure 2.7.: *Conceptual view of a typical* TTCN-3 *test configuration*

between test components and the test system is also done via ports. Beside, the *Real Test System Interface* (RTSI) defines the implementation dependent adaptation between ATSI and SUT (see Figure 2.7). The access respectively communication points between ATS and SUT were called PCO in TTCN-2.

Test case execution is defined in the module control part and may be structured by functions. A **testcase** is used to initiate a test case where the MTC is automatically created. Using the **runs on** and **system** clause for **testcase** the MTC and ATSI can be given. Functions are used to structure PTC behaviour. Some available operations are

- **create** to create a component,
- **connect** to connect ports between components,
- **map** to connect ports between components and the test system interface,
- **running** to check if a component is still running, and
- **done** to wait for the termination of a component.

In Figure 2.5, 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

Listing 2.5: TTCN-3 *Test configuration – Component type definition*

```
1  type component MainTC {
2    port InitiatorSAP ISAP1;
3    port PortAtMTC CoordinationPTC;
4    timer supervisionTimer;
5  }
6
7  type component ParallelTC {
8    port MediumSAP MSAP2;
9    port PortAtPTC CoordinationMTC;
10 }
11
12 type component TestSystem {
13   port InitiatorSAP ISAP1;
14   port MediumSAP MSAP2;
15 }
```

MSAP2, is mapped to a port with the same name of the ATSI. 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 realised by a RTSI which is outside the scope of TTCN-3 (see Figure 2.7).

In Listing 2.5, TTCN-3 test component definitions are presented 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).

2.4.6. Behaviour Description

Functional and dynamic behaviour of components over ports is described by functions, test cases, alternatives, and altsteps and executed in module control parts. Test cases are used for the behaviour description of MTCs. Functions are used to structure and specify test behaviour, to organise test execution, and to structure computation in a module. For instance, functions can be used to organise behaviour of PTCs.

In the following these concepts are described in more detail. Examples of a test case, function, and altstep definition are given in Listings 2.6, 2.7, and 2.8.

Listing 2.6: TTCN-3 *Test case* SingleDataTransfer

```
1  testcase SingleDataTransfer() runs on MainTC system TestSystem {
2    var ParallelTC ptc;
3    var default def1, def2;
4
5    ptc := ParallelTC.create;
6
7    map( self:ISAP1, system:ISAP1 );
8    map( ptc:MSAP2, system:MSAP2 );
9
10   connect( self:CoordinationPTC, ptc:CoordinationMTC );
11
12   ptc.start( MediumAccess() );
13
14   def1 := activate( MTCFailure() );
15   def2 := activate( ReceptionIDISind( inconc ) );
16
17   ISAP1.send( ICONreq : {} );          // connection request
18   ISAP1.receive( ICONconf : {} );      // connection confirmation
19
20   supervisionTimer.start( maxTransferTime );   // restrict time for data transfer
21
22   ISAP1.send( InresDataRequest( someUserPDU ) );   // data transfer
23
24   // delay disconnection request until 'ptc' has received and acknowledged the data
25   CoordinationPTC.getcall( acknowledgementSent : {} );
26   CoordinationPTC.reply( acknowledgementSent : {} );
27
28   supervisionTimer.stop;    // cancel timer to avoid a timeout in the following
29
30   deactivate( def2 );    // a disconnection indication is no undesirable event any longer
31
32   ISAP1.send( IDISreq : {} );          // disconnection request
33   ISAP1.receive( IDISind : {} );       // disconnection indication
34
35   all component.done;
36
37   setverdict( pass );
38 }
```

Control Structure

Behaviour is described in sequential order by basic control structures like **while**, **for**, **do...while**, and **if...else** and alternative behaviour statements like **alt** and **interleave**. Furthermore, usage of **goto** with labels is possible to support conversion of TTCN-2 test cases to TTCN-3.

The **alt** statement describes a set of alternative behaviour depending on the reception and handling of communication, timer events, and PTC termination which are called *reception statements*. Each alternative consists of the three parts boolean expression, guard, and statement block. The statement block of an alternative gets executed if the corresponding guard

can be executed which has to be a reception statement. The boolean expression has to evaluate to true or has to be empty to enable a guard. It is possible to provide as last alternative an optional **else** part which gets executed if no alternative matched.

To evaluate guards a snapshot semantics is defined. Entering an alternative statement activates the snapshot mechanism where the state of all relevant components, port queues, and timers are frozen to allow undisturbed evaluation of the boolean expressions and guards. The evaluation order is given by the order of the alternatives. If no guard can be executed, a new snapshot is taken and evaluation starts from the beginning until one guard can be executed. However, an alternative is called blocked if all guards can never be fulfilled which must lead to a test error. There is the **repeat** statement to initiate a re-evaluation of an **alt** statement where a new snapshot is taken and evaluation starts again from the first alternative.

If exact order of receiving messages or procedures is not predictable or not important, an interleaved handling is possible by the **interleave** statement. Thus, writing down all possible combinations of alternatives is not necessary. Interleaves are structured like alternatives with empty boolean expressions and with no else clause. Furthermore, usage of control statements, functions, and communication operations inside interleaves is restricted.

Altsteps

Beside structuring behaviour by functions it is especially possible to structure **alt** statements by usage of **altstep** where a collection of alternatives can be defined. Additionally, default behaviour can be defined via altsteps. Altsteps provide local definitions and own parameters including a **runs on** clause. The body of altsteps is structured by a set of alternatives like in **alt** statements.

There is a default mechanism for alternatives in TTCN-3 where altsteps can be activated by default. Defaults are stored in a default list and can be activated and deactivated any time by operations **activate** and **deactivate** respectively. The **deactivate** operation requires a default reference which is delivered by the **activate** operation and which can be stored in a **default** type. A default reference gets necessary because an altstep can be activated with different parameters. The default list is called if no alternative of an **alt** statement can be executed. The default altsteps and their alternatives are evaluated step by step until an alternative can be executed. If

Listing 2.7: TTCN-3 *Function* MediumAccess

```
1  function MediumAccess() runs on ParallelTC {
2    var integer     receipt;
3    var default     def := activate( PTCFailure() );
4    var MDATind  indication;
5
6    MSAP2.receive( ConnectionRequest );
7    receipt := 1;    // first (received) connection request of the initiator
8    MSAP2.send( MediumDataRequest( ConnectionConfirmation ) );
9
10   alt {
11     [ receipt <= maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
12         receipt := receipt + 1;
13         MSAP2.send( MediumDataRequest( { CC, omit, omit } ) );
14         repeat;
15       }
16     [ receipt > maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
17         setverdict( fail );
18         stop;
19       }
20     [] MSAP2.receive( DataTransfer( someUserPDU ) ) -> value indication { /* empty */ }
21   }
22
23   MSAP2.send( DataAcknowledgement( indication.mSDU.seqNo ) );
24
25   CoordinationMTC.call( acknowledgementSent : {} );
26   CoordinationMTC.getreply( acknowledgementSent : {} );
27
28   MSAP2.receive( MDATind : { mSDU := { DR, omit, omit } } );
29   setverdict( pass );          // disconnection request
30 }
```

there is no executable alternative found in the default list then the default mechanism will return back where it has been invoked.

In Listing 2.8, altsteps *MTCFailure* and *ReceptionIDISind* are defined. Altstep *MTCFailure* makes test execution fail if a message is received at port *ISAP1* that is not handled elsewhere or a timer expires. Altstep *ReceptionIDISind* illustrates the definition of a parameterised altstep. Depending on variable *result*, the reception of message *IDISind* leads to different test verdicts.

Timer

TTCN-3 supports usage of timers. Local timers declared in component type definitions are running on the component. There are operations to control and evaluate timers. The **start** and **stop** operation are used for control of timers. The elapsed time after starting a timer can be retrieved by the **read** operation. The status of a timer can be get by the **running**

Listing 2.8: TTCN-3 *Altsteps* MTCFailure *and* ReceptionIDISind

```
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
12 altstep ReceptionIDISind( verdicttype result ) runs on MainTC {
13   [] ISAP1.receive( IDISind : {} ) {
14        setverdict( result );
15        stop;
16      }
17 }
```

operation which delivers a boolean to express if the timer is still running. The **timeout** operation can be used in guards of alternatives to check if a timer has expired or is used as stand-alone operation. A stand-alone **timeout** operation blocks execution until the timer expires.

Verdicts

The TTCN-3 special basic type **verdicttype** is used to handle test verdicts. Only the five distinguished values **pass**, **fail**, **inconc**, **none**, and **error** are available as verdicts. Test verdicts are delivered by test cases and depend on the local verdict of each component of a test case. After termination of a component the test case verdict gets updated. Component verdicts can be set and read by the **setverdict** and **getverdict** operations respectively. Verdicts can only be downgraded why it is not permitted to revise a verdict from **fail** to **pass**, for instance. Test verdict setting does not stop test case execution.

2.5. Summary

In the first section of this chapter dynamic testing concepts have been classified to provide an overview and motivation about different kinds of dynamic tests. The classification distinguishes between type of implementation, phase in development cycle, system knowledge, test objectives, test data selection, and authority for test results.

The second section explained why testing object-oriented systems is different from testing of other systems and why static testing is not sufficient. The object-oriented concepts encapsulation, inheritance, and polymorphism as well as method interaction and language and compiler services make testing more difficult.

Thirdly, manual and automatic test generation were discussed. Some remarks about the test generation tools AUTOLINK and TESTCOMPOSER were given.

Fourthly, the test description language TTCN-3 was described which permits the specification and implementation of tests for distributed systems in an implementation language independent manner. Important to mention is the possibility of importing data specifications from other languages, the introduction of procedure-based communication, and dynamic test configuration.

3. UML-based Testing

As explained in the preceding chapter, testing requires automation and interworking between system and test development. Nowadays, many (object-oriented) systems are described using the UML which is defined by the OMG (OMG 2003c). Thus, UML models are an important source for test development why using UML from a test perspective has to be considered. Additionally, there is ongoing work on an UML *Testing Profile* (UTP) (Schieferdecker et al. 2003) and an improved version of UML, namely *Unified Modeling Language 2.0* (UML 2.0) (Jeckle et al. 2004; OMG 2003b).

The automatic test generation process approach based on SDL, MSC, and TTCN as described in section 2.3 is used as inspiration for UML-based scenario testing. Scenario-based testing, manual as automatic, is applicable for black-box and specific white-box testing for communication protocols and distributed systems like CORBA-based systems. In particular, manual scenario-based testing is often used by test designers to test specific parts of the SUT which are interesting or not covered by an automatic test generation process. Furthermore, there is a discussion of TTCN-3 *Graphical Presentation Format* (GFT) in context of MSC and UML in Schieferdecker & Grabowski (2003) available where the close relation between all three standards is detailed.

This chapter is structured as follows. Firstly, some remarks to UML and its diagram types are given. Secondly, suitability of UML models for testing is discussed. Thirdly, scenario-based testing with UML is explained. Fourthly, MSCs are explained. Fifthly, the IDL will be described. Finally, a summary and an outlook are given.

3.1. Unified Modeling Language

Unified Modeling Language (UML) is a model with focus on structure and behaviour definition for object-oriented systems for which several diagram types are defined (OMG 2003c). It defines only the notational syntax and informal semantics for object-oriented models but there is no methodology given. Some model elements can be used in several diagrams. Each

diagram has its own usage and point of view on the system why the system can be described from many different perspectives. Hence, system developers can choose the right view to describe a specific system structure or behaviour. There are several tools which support drawing UML models and code generation for different languages like C++, JAVA, and IDL.

The following diagram types are available in UML version1.5:

Usecase diagrams are used to describe in an abstract way the system response to external inputs given by some external actor, normally humans. Mainly used for specifying system requirements.

Class diagrams describe the class or interface structure including their attributes, methods, and relationships. Class diagrams are used to describe distribution of data and behaviour to classes.

Behaviour diagram types are used for describing dynamic issues.

- *Activity diagrams* or object flow charts show sequences of activities (processes not states) where concurrent execution is possible. They are mainly used to model human work flow and may be associated to a usecase or class diagram.
- *Collaboration diagrams* describe interactions among objects in context or a limited role under emphasis of relationship between objects and their topography. Mainly used to explain or to document sequences.
- *Sequence diagrams* are used to describe sequences of message exchanges between objects in a time limited situation with emphasis on the temporal order. Can be used to describe necessary collaborations to implement a use case.
- *State diagrams* show the sequential control requirements of objects by sequences of states depending on external stimuli (finite state machines).

Implementation diagram types are used to describe implementation issues.

- *Component diagrams* show dependency relationships between components which have their own identity and a defined interface. Components define boundaries and groups and organise elements.

Deployment diagrams represent objects and components on nodes like hardware, software, or network architecture and their communication associations. Deployment diagrams are useful for integration planning.

Sequence diagrams and collaboration diagrams are describing the same issue. Contrary to sequence diagrams, where the temporal order is the focus, collaboration diagrams are focussed on the working relationship. Diagrams can be grouped and organised into hierarchical packages where package dependencies can be described. This kind of diagram is realised as a special case of class diagrams.

3.2. Suitability of UML for Testing

Using a system specification defined in UML to generate, maybe automatically, and to specify test cases the applicability of UML and its diagram types has to be inspected first (Binder 2000, chapter 8).

Using UML as a test model requires unambiguousness to support automatic production and relation to probable faults to provide a test design why identification, analysis, and demonstration of relationship is required. UML provides only a notational syntax where no prescription of result, technique, or process is given. Furthermore, UML is very flexible because only restrictions on usage of notation are given. However, models can be fragmentary, incomplete, inconsistent, and ambiguous without violating UML. Additionally, the models are indifferent to testability because combinational logic and domain definitions are missing which requires much hand work to produce tests. Thus, automatic test code generation is not possible. Nevertheless, the models are still an important source for test development. Responsibility and architecture of SUT can be modelled and partly automatic test generation may be possible if models are complete, correct, and consistent.

Usecase diagrams use input and output variables without providing any definition and conditions are missing to determine basic and alternate flows (Miga et al. 2001; Mulvihill 2003). They can be instantly mapped onto sequence diagrams. However, usecase testing alone is not enough because of incomplete system coverage. Class diagrams are the main resource for structural information like types, methods, and attributes and therefore, are very important for test specification. For instance, information missing in other diagram types like usecases can be obtained from class diagrams. Furthermore, test cases could be generated to test mutual

constraints and dependencies defined by associations, correct create and destroy of aggregated objects, and correct usage of generalisation. Activity diagrams could be used to develop test models by control flow where, for instance, decision tables and composite control flow graphs for a collection of sequence diagrams are usable. Control flow graphs at method state could be used for analysing path coverage. Furthermore, structuring test cases can be modelled by activity diagrams, especially in conjunction with associated usecase or class diagrams.

Collaboration diagrams show all required methods for a complete interaction. They are usable for structuring an implementation and for specifying a method implementation, all methods in a class, or an usecase. Only a small slice in comparison to the whole system is described wherefore coverage analysis is limited to this slice. The method call hierarchy must not be a tree why it is ambiguous. However, transformation into several sequence diagrams with resolved ambiguities is possible. Sequence diagrams show how the collaboration via message exchange is done in the temporal dimension to implement, for instance, a usecase. A sequence diagram cannot describe all possible paths why several sequence diagrams are necessary to get all paths. Sequence diagrams are less expressive in the selection and iteration notation, the conditional and delayed message distinction is weak, and dynamic binding and unique superclass/subclass behaviour cannot be shown. Using the *round-trip scenario* test pattern a sequence diagram is used as test purpose where each possible path can be used for test case generation. The intention of the test approach is to "extract a control flow model from a UML Sequence Diagram and develop a path set that provides minimal branch and loop coverage" (Binder 2000, page 579). State diagrams describe possible states and transitions of objects by usage of finite state machines but the provided notation is less expressive and lacks a well defined semantic. Thus, state diagrams are very important for automatic test generation by coverage analysis but due to their shortcomings usage for testing is difficult. However, using usecase and sequence diagrams could partially replace state-based testing. In (Binder 2000, chapter 7) usage of state models like UML state charts, *Object Modeling Technique* (OMT) dynamic model, and *Real-time Object-Oriented Modeling* (ROOM) statechart are discussed to test object-oriented systems. In Mellor & Balcer (2002), using executable UML is described.

Component diagrams describe distribution and dependencies of components with own interfaces (implementation entities) like classes from a physical point of view which is used, for instance, to describe concrete distribution into libraries. Thus, component diagrams provide structural

information which can be used to identify method call paths. The paths found can be expressed in sequence diagrams to allow test case specification and generation. Deployment diagrams represent distribution of objects and components on nodes and communication associations between nodes. Nodes are used for execution. Depending on the number of shown details in deployment diagrams they can be used for dependency analysis and for integration planning especially.

Hence, existing diagrams from the specification can be used directly for test generation such as sequence diagrams, if fully defined. Additional diagrams, especially for scenario-based testing, can be specified, too (Amyot & Eberlein 2003). Thus, automatic and manual test case generation and specification are together possible. Test case generation based on test purposes specified by sequence diagrams where each possible complete path gets a test case is usable, but usage of test coverage analysis is difficult. Overall, UML is missing a well defined semantics to be well applicable for automatic test generation (Binder 2000, chapter 18). See Binder (2000, chapter 9) for test strategies and test patterns. A coverage model for object-oriented systems is given in Binder (2000, section 4.4).

3.3. UML-based Test Specification

As discussed in the section before, existing system specifications done via UML could be used for test generation. However, some diagram types could be used for test purpose or test case specification especially. Thus, scenario-based, manual test specification with UML is more interesting where the test designer can focus on specific elements in the SUT and requires no test specific knowledge like the used test language.

One diagram type does not provide enough information to specify and generate a complete test case, and responsibility and architecture of a SUT can only be modelled if the used models are complete, correct, and consistent. However, if we take a look into test suites defined via TTCN-3 we can find a partitioning into a statical and dynamic part. The statical part contains elements like type, test data, method signature, and test configuration definitions, whereas the dynamical part contains elements like method calls and their order, response evaluation, test configuration set up, and test result handling. Thus, usage of class diagrams for static information and sequence diagrams for dynamic information to generate test cases is quite enough. Furthermore, other UML diagrams can be used to generate class or sequence diagrams (see Figure 3.1).

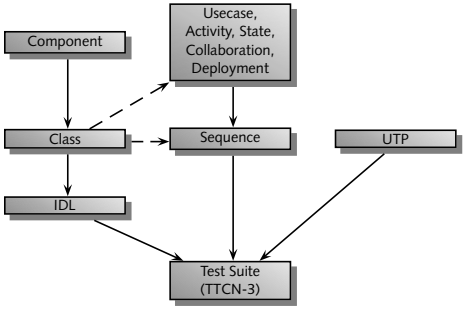


Figure 3.1.: UML-*based test specification*

Information from component diagrams can be translated into class diagrams and usecase, activity, state, and collaboration diagrams can be used to generate sequence diagrams. More information about generation of sequence diagrams are given in Binder (2000). For instance, activity diagrams are associated to class and usecase diagrams, a usecase or collaboration diagram can be used instantly to generate several sequence diagrams. Activity diagrams can also be used to control test case execution order as can be done with *High-Level Message Sequence Charts* (HMSCs) (see section 4.5).

The concept of using static and dynamic information from different sources and its applicability and usefulness was shown earlier by usage of SDL as static source and MSC as dynamic source as described in section 2.3. SDL is a very expressive state-based specification language and MSCs are used to describe message exchanges like in sequence diagrams. Despite its good results in testing communication protocols, tool vendors like Telelogic with ist tool TAU G2 shift from SDL and MSC to UML to reach a wider audience. Due to this shift a combination of SDL and MSC with UML or at least the integration of well-defined and widely used concepts of SDL and MSC into UML is wished to enhance UML with more exact semantics. Thus, automatic test generation from UML would be more expressive. Additionally, there is ongoing work to define the UML *Testing Profile* (UTP) which allows test suite specification and presentation inside UML (Schieferdecker et al. 2003). There is ongoing work on ITU – *Telecommunications Standardisation Sector* (ITU-T) standard Z.149 to define a mapping of UTP to TTCN-3. Hence, UTP specifications could be used to provide a more complete test suite.

TTCN-3 is a well established testing environment and therefore, usage of TTCN-3 for test suite specification is used in this thesis where only class and sequence diagrams have to be mapped to TTCN-3. Missing parts to provide a full TTCN-3 test suite can be integrated by using UTP, and generated test cases from class and sequence diagrams may be used in UTP, too. Most UML tools support conversion from class diagrams into IDL where all required information for test generation are still available. Hereby, IDL can be used instead of class diagrams to widen application (see Figure 3.1). Mapping sequence diagrams and IDL to TTCN-3 is described in chapter 4 and chapter 5 respectively.

3.4. Message Sequence Chart

Sequence charts like UML sequence diagrams (OMG 2003c) and *Message Sequence Charts* (MSCs) (ITU-T 2001) are used to specify and describe communication behaviour by message interchange including procedure calls between several distributed entities in a temporal order. Since UML sequence diagrams do not provide well defined semantics in contrary to MSCs, MSCs are more expressive than sequence diagrams. MSCs are well used for test purpose and test case specification and MSCs support most functionality of sequence diagrams, so MSCs are used to discuss usage of sequence charts to specify test cases (see chapter 4). Furthermore, there is ongoing work which introduces MSC concepts into UML 2.0 which leads to the convergence of sequence diagrams to MSCs (Jeckle et al. 2004; OMG 2003b).

MSC is a graphical specification language standardised by ITU-T as Recommendation Z.120 (ITU-T 2001). It is a scenario language to describe communication behaviour between system entities and their environment. There is a textual and a graphical notation available whereat the textual notation is mainly used by tools to store and interchange charts and the graphical notation is used for intuitive representation of charts. Three types of charts are provided by MSC. Namely, (basic) MSC to describe concrete events in a temporal ordering, *High-Level Message Sequence Chart* (HMSC) to illustrate how to combine MSCs, and MSC documents which serve as a summary of belonging charts. The current version of MSC is called *Message Sequence Chart-2000* (MSC-2000) (ITU-T 2001) and provides better support, in comparison to its predecessor MSC-96 (ITU-T 1996), for real-time systems, object-oriented systems, and sequen-

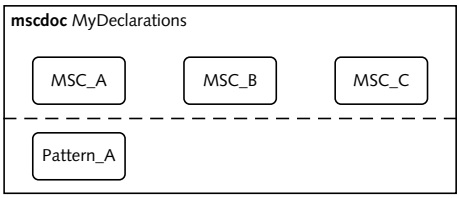


Figure 3.2.: MSC *document example*

tial programs by added concepts for data, time, control flow, and object-orientation on MSC documents.

In the following MSC-2000 will be detailed by document structure, basic elements, structural elements, and HMSCs.

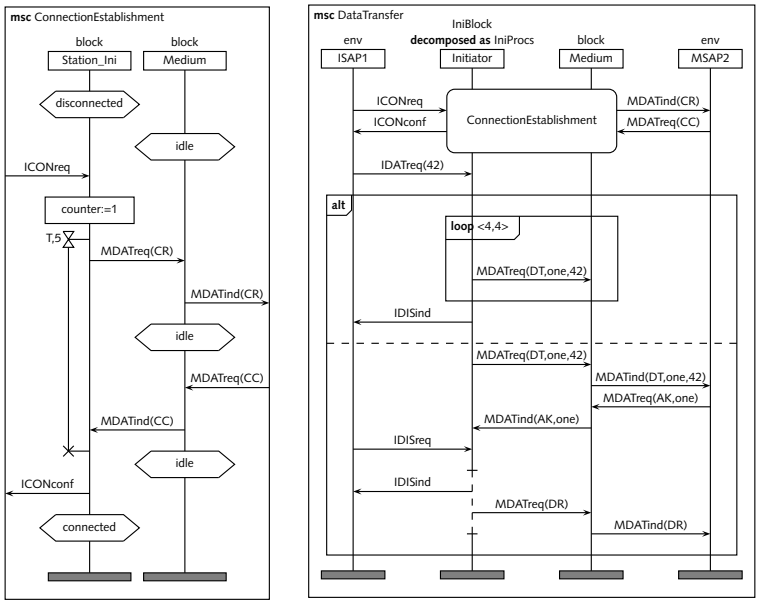
3.4.1. MSC Documents and Comments

MSC documents are used to provide an associated collection of MSCs (set of traces) and define all kinds of instances used in the MSCs. In the *definition part*, available MSCs are defined, and the *utility part* is used to describe patterns of MSCs which can be used by MSCs in the definition part (see Figure 3.2). The relation to documents like TTCN, SDL, and UML documents can be given by the keyword **related to**. Instance definitions can use inheritance concepts to describe among others decomposition of instances.

MSC supports three kinds of comments. The *note* is only available in textual notation, the *comment* is used for informal explanations associated with symbols or text, and the *text* is used for global comments associated to a chart.

3.4.2. Basic Message Sequence Charts

Basic MSCs are used to specify and describe the communication flow between system entities where the concept of *instances* and *messages* is used. Communication with the *environment* can be described and usage of *gates* to compose MSCs is supported. *Actions* are used to describe internal behaviour of entities and *conditions* are used to restrict number of traces. *Timers* are available to express time limits for execution. Instances can be dynamically created and terminated. Basic MSCs are now explained in detail.



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

Figure 3.3.: *Basic* MSC*s for the Inres protocol*

In Figure 3.3(a) an MSC with two instances, *Station_Ini* and *Medium*, is shown 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 (see subsection 2.4.1). 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 de-

scription of an instance finishes with a special end symbol. However, this symbol does not mean that the instance actually terminates.

Instances, Messages, and Control Flow

The base elements of basic MSCs are instances and messages. Instances represent system entities which exchange messages with each other and the environment. Instances have a name and can have a type, and messages have a name and optional parameters. Message interchanges are divided into asynchronous events, where **in** events are used for receiving a message and **out** events for sending a message. If there is only an **in** event available an unknown sender message can be described and if there is only an **out** event used a lost message is described. Messages have an optional time attribute wherewith delay of message sending and receiving can be specified. The textual representation of MSCs may be ordered by instances or events where an event order describes a valid execution trace.

Temporal ordering of exchanged messages is defined by a total temporal ordering on instance axis and a partial order between instances where only the order is defined but no concrete timing. Hence, it is nonrelevant if a message arrow is directed up or down because a message has to be send first to get consumed. However, in case where unrelated events on different instances have to be ordered a *general ordering* can be defined explicitly. Furthermore, in case ordering on a instance axis is not important it can be suspend by *coregions* for a part of the axis. Nevertheless, the usage of general ordering in coregions is possible.

Apart from message interchange, MSC supports synchronous message exchange by means of calls and replies. Calls are like remote method invocations where the result is returned by the reply. The call name represents the named unit of behaviour inside an instance. The reply uses the same name as the corresponding call. Calls can be distinguished into blocking and non-blocking calls. In case of a blocking call the caller enters a suspension region where the caller is waiting for a reply and no other events may happen, whereas in case of a non-blocking call the caller may proceed further. Non-blocking calls are called asynchronous whereas blocking calls are called synchronous.

Environment and Gates

The environment is represented by the rectangular frame of each MSC (see Figure 3.3(a)). Communication between instances and environment

is permitted by message interchange. Each **in** and **out** message with the environment is assigned to a *gate* (interface to environment) which allows to compose MSCs in conjunction with usage of MSC references inside an MSC. The environment provides no total ordering for messages why instances should be used as environment if gates are not required or ordering is required.

Actions

Beside message exchange, internal actions of instances can be given by informal text or formal data statements which can be used to modify internal states or counters, for instance. Actions are an atomic element and are represented by a rectangular symbol. For instance, in Figure 3.3(a) the variable *counter* of instance *Station_Ini* is set to one.

Conditions

Conditions are used to restrict valid traces or the composition of MSC references in HMSCs. Conditions can be used for one, several, or all instances. There are two kinds of conditions, namely *setting* and *guarding* conditions. A setting condition defines a state of an instance where the state is given by a state name. If all instances are involved, it describes a global state. A guarding condition, depending on a boolean expression, restricts the following event execution. The boolean expression can be given in the data language or by an active state which was set in a former condition.

Timers

Beside the temporal order of messages, the usage of timers is possible to express duration dependencies (see Figure 3.3(a)). Timers are described by the three events **start**, **timeout**, and **stop** and corresponding events have to be attached on the same instance. Timer duration for timeout is settable by lower and upper bounds. There is a global clock assumed but global timers are not supported.

Instance Creation and Termination

Apart from static creation, instances can be dynamically created by other instances. Instance termination is done by instances itself. Instance creation is represented by a dashed arrow beginning at the creating instance

and ending at the new instance head. Instance termination is represented by a cross. Since all instances, static and dynamic, have to be explicitly shown, the creation of an unknown number of instances is not possible. This may be a hard restriction but it is sufficient for scenario description which is the purpose of MSCs.

3.4.3. Structural Concepts

Beside basic MSCs the structural concepts *coregions*, MSC *references*, *instance decomposition*, and *inline expressions* are supported which will be described in the following.

Coregion

Total ordering on instance axes can be suspended by *coregions* where ordering gets changed to allow any event order. For instance, sending or receiving several messages may be interchanged. Suspending total event ordering can be useful to describe higher-level systems, too. Ordering in coregions can still be ordered or limited respectively by usage of the general ordering mechanisms. Coregions are graphically marked by a dashed instance axis as done in Figure 3.3(b) on instance *Initiator*.

Inline Expressions

Inline operators are used for easier definition of event structures. The operators **alt**, **par**, **seq**, **opt**, **exc**, and **loop** are available to define alternative, parallel, and sequential composition, optional regions, exceptions, and iterations.

Alternative composition is used to define alternative execution traces of an MSC whereby only one trace gets executed. The choice between different traces has to be done after executing the common part of the possible traces. Parallel composition represents parallel execution of all defined sections whereas event order within each section gets preserved. Sequential composition represents the weak sequencing operation. Optional events can be defined by an optional region which is equal to an alternative with an empty MSC as second operand. Exceptional cases can be handled by the exception operator which is equal to an alternative with the remaining MSC as second operator. Thus, if the exception part gets executed the MSC terminates. The option and exception operator are very similar except continuation or termination of the MSC after operator execution. As last operator the **loop** operator is provided to define iterations.

Number of loop executions are defined by a lower and upper boundary in which usage of the keyword **inf** is supported to express an infinite boundary.

An alternative and loop expression is shown in (see Figure 3.3(b)).

MSC References

Due to structure MSCs an MSC can reference other MSCs of the MSC document. All instances used in a referenced MSC must also appear in the calling MSC. Events can be send to and received from a referenced MSC by gates. References may have parameters which have to match with the corresponding MSC parameter declaration. MSC reference expressions can be defined by the operators **alt**, **par**, **seq**, **loop**, **opt**, and **exc** as defined by inline expressions before. Therefore, several MSCs can be referenced.

At the beginning in Figure 3.3(b) a reference to MSC *ConnectionEstablishment* is given.

Instance Decomposition

In addition to MSC references instances can be decomposed to structure charts as done with instance *Initiator* in Figure 3.3(b). Instance decomposition can be used to control level of detail where interaction is described. Behaviour and internal structure of decomposed instances are defined by an own MSC document which uses the instance kind name as document name. The behaviour is described by an MSC and the structure by used instances. A hierarchy of decomposed instances may be used.

3.4.4. High-Level Message Sequence Charts

Apart from MSC documents and basic MSCs there are *High-Level Message Sequence Charts* (HMSCs) available as another structuring type. They are directed graphs which describe how to combine a set of MSCs. Thus, HMSCs provide a higher description and structuring level by abstracting from concrete message exchanges.

Each node in an HMSC can be either a start or end node, an MSC reference, a condition, a connection point, or a parallel frame. Nodes are connected by flow lines and flow lines can be connected by connection points. MSC references may use reference expressions to reference to several charts. Conditions are used for describing system states, guards, or restrictions as similarly done by conditions in basic MSCs. Parallel frames

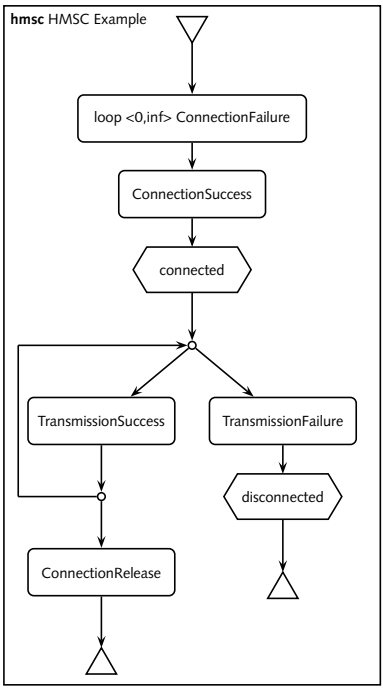


Figure 3.4.: HMSC *example*

contain smaller HMSCs which are part of a parallel operator and hence, are executed in parallel and events from different smaller HMSCs can be interleaved. An HMSC example is given in Figure 3.4.

3.5. Interface Definition Language

There exist several standards which define an *Interface Definition Language* (IDL). However, the IDL standard defined and used in the CORBA is the most popular one and others are mostly derived from it (ISO/IEC 1999; ITU-T 1997a; OMG 2001b). Furthermore, we are in particular interested in using it for CORBA-based systems and if the IDL standard is imprecise or lacking some information, the CORBA standard can be used as reference. Therefore, only CORBA IDL is mentioned here.

The IDL is a language to *describe* interfaces in an implementation language independent manner and can also be used by other systems than CORBA. It does not support the description of implementation characteristics like behaviour, instances, or relationships.

For better understanding of IDL, the CORBA gets explained first. Afterwards, IDL is explained in detail by the object model, data types, modules and interfaces, and attributes and operations.

3.5.1. Common Object Request Broker Architecture

The *Common Object Request Broker Architecture* (CORBA), defined by the OMG, is a standard architecture for distributed object systems. The heart of CORBA is the *Object Request Broker* (ORB) which is the communication infrastructure for the distributed environment. The ORB provides a mechanism for transparently communicating client requests to target object implementations. It simplifies distributed programming by decoupling the client from the details of the method invocations, and hence, makes client requests appear to be local procedure calls. The ORB consists of the ORB core and some interfaces on top of it. The ORB core provides the basic representation of objects and means for the communication of requests.

The technology used in the ORB core is hidden by the public interfaces layered on top of it. There are the IDL *Stub* and *Skeletons* which present the language mapping and support the static invocation of requests to objects, the *Dynamic Invocation Interface* (DII) and the *Dynamic Skeleton Interface* (DSI) which allow dynamic creation and invocation of requests to objects at run-time, and the *Interface Repository* (IR) that provides storage of object interface definitions which are accessible by applications at run-time (see Figure 3.5).

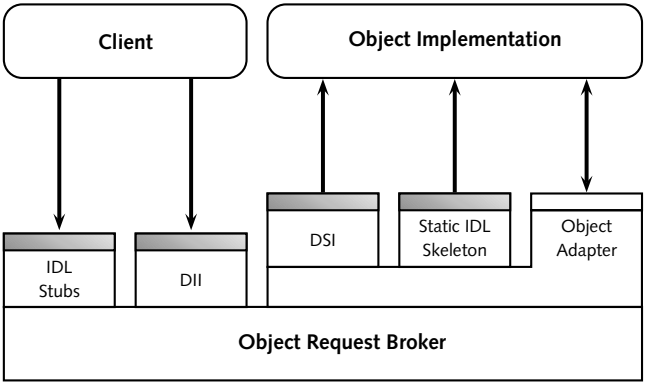


Figure 3.5.: *The* CORBA *architecture*

3.5.2. Object Model

To understand IDL, one must know the *object model* of CORBA. All features of the object model must be captured by IDL because it shall describe these features. Therefore, the object model and IDL can be described together. All that is valid for the object model should be valid for IDL and vice versa except the special restrictions of a model and a language. An example for this exceptions is the module and data type concept of IDL.

The object model of CORBA is the elementary part of the whole standard. It describes the view, called *interface*, on each object. This view is described using the IDL of the OMG for CORBA (OMG 2001b, chapter 3). It is a language to *describe* interfaces in an implementation language independent manner. It shall not describe implementation characteristics like behaviour, instances, or relationships. Therefore, the following constructs are defined:

Constants to assist with type declarations,

Data type declarations to use for parameter typing,

Attributes which allow getting and setting of a value of a particular type,

Operations which take parameters and return values,

Interfaces which group data type, attribute, and operation declarations,

Modules for name space separation.

Thus, each *client* can invoke remote operations on an object without knowing about the implementation details. The communication between objects per default is peer-to-peer and synchronous in an *at-most-once* manner. Asynchronous communication is only available in using *one-way* and *best-effort* manner.

The IDL is a base of the whole CORBA standard and an important point in developing distributed systems with CORBA. It allows the reuse and interoperability of objects in a system. A mapping between IDL and a programming language is defined in the CORBA standard. IDL is very similar to C++ in containing pre-processor directives (include, comments, etc.), grammar as well as constant, type, and operation declarations. There are no programming language features like, e.g., if-statements.

The most important part of IDL are *operation* specifications. The operations are a part of *interfaces* for an object. Interfaces may be summarised to *modules*. This leads to hierarchical composition with naming scopes. The next sections explain the data types, module and interface concept, and operation declarations.

3.5.3. Data Types

IDL supports the most basic data types from C++, but there are no *references* in IDL. Instead, the types **string**, **boolean**, and **any** are available. Some data types have a different specification, like **char** which is a type of its own in CORBA. The *constructed* types **enum**, **struct**, **array**, and **union** are similar to the ones in C++. The *template* type **sequence** is a variable length array of elements of one, but any, IDL type. The type **string** is like **sequence** but only supports ASCII ISO-LATIN characters. Type **any** is like type **Object** in JAVA a placeholder for any possible IDL type (see Table 3.1). A code example is given in Listing 3.1.

3.5.4. Modules and Interfaces

The main concept behind IDL consists of interfaces. Each interface may contain constants, types, attributes, exceptions, and operations for one object. A module is a method to separate name spaces and may contain any IDL construct. Each IDL construct is automatically *public* according to the object-orientated concept. (Multiple-) Inheritance is only permitted for interfaces, not for modules. Forward declarations of interfaces permit their use before their definition. The concepts can be seen in Listing 3.2 which illustrates an interface hierarchy structure.

Table 3.1.: *Overview of* IDL *types*

Class of Type	*Type (Keyword)*
Basic	short, long
	long long
	unsigned short
	unsigned long
	unsigned long long
	float, double
	long double
	char, wchar
	boolean
	octet
	any
Object reference	Object
Constructed	struct
	union
	enum
Template	sequence
	string, wstring
	fixed
Complex	arrays
Native	native

Concrete language mappings between C, C++, SmallTalk, Cobol, Ada, Java, and IDL can be found in the CORBA standard (OMG 2001b).

3.5.5. Attributes and Operations

Each client knows the IDL interface specification of each *object* containing all information about the *object*. Attributes are like variable definitions but they behave like operations in CORBA. Each *read-write* attribute gets a *set-* and a *get*-function and each *read-only* attribute gets a *get*-function. The main part of an interface is based on operations. An operation declaration consists of

- an operation attribute that specifies the invocation semantics,
- the type of the operation result,
- the operation name,
- a parameter list,
- optional exceptions, and
- optional context expressions.

The default operation attribute is synchronous invocation and *one-way* stands for asynchronous invocation. Each operation may return exactly one value or must be *void*. The parameter list contains the direction, type,

Listing 3.1: *IDL data type example*

```
1   const long    number    = 017;  // 017 == 0xF == 15
2   const float   decimal   = 15.7;
3   const char    letter    = 'A';
4   const boolean isValid   = TRUE;
5   const octet   anOctet   = 0x55; // limited to 8 bit
6   const string  myName    = "my name";
7
8   union MyUnion switch( long ) {
9       case 0 : boolean b;   case 1 : char c;   case 2 : octet o;      case 3 : short s;
10  };
11
12  enum NotFoundReason { missing_node, not_context, not_object };
13
14  typedef sequence <NameComponent> Key;
15  typedef fixed<12,7> Fix;
16  typedef long NumberList[100];
17  typedef struct  NC { MyString id;  MyString kind;  }  NameComponent;
```

Listing 3.2: *IDL structure example*

```
1  module InheritanceExample {
2      interface  A { ... };
3
4      interface  B : A { ... };
5  };
```

and name of each operation parameter (see Listing 3.3). The direction information tells the direction in which the parameter is to be passed where the following directions are supported:

in to pass the parameter from client to server,

out to pass the parameter from server to client, and

inout to pass the parameter in both directions.

Listing 3.3: *Structure of an IDL operation declaration*

```
1  returnValue  operationName( in Type1 par1, inout Type2 par2, out Type3 par3 )
2      raises (  AnException )
3      context (  "ContextInformation" );
```

Listing 3.4: *IDL interface example*

```
1  interface NamingContext {
2      attribute string object_type;
3      readonly attribute Key external_form_id;
4
5      exception NotFound { NotFoundReason why; Name rest_of_name; };
6
7      MyString bind( in Name n, inout Object obj, out Object myObj )
8          raises( NotFound )
9          context ( "Hostname" );
10
11     oneway void rebind( in Name n, in Object obj );
12 };
```

Exceptions are especially used to handle errors caused by the network environment like connection failures. The context expression allows the client to transfer context specific information, like security context information for the security service.

An example containing declarations of interfaces with operations and attributes is given in Listing 3.4. It was taken from the *NamingService* of the *CORBAservices* (OMG 1997).

3.6. Summary and Outlook

Summary

In this chapter first, the different diagram types of UML were shortly explained. Afterwards, suitability of UML models for automatic test generation and usage for test case specification were discussed. UML models until now are not suitable for automatic test generation because of shortcomings in its semantics and missing features as described in section 3.2 (Binder 2000).

Test case specification is possible wherefore MSCs, as substitute for sequence diagrams, are best suitable from all UML diagrams. Some other diagram types, for instance usecase diagrams, can be mapped to MSCs as discussed in section 3.3 (Miga et al. 2001). In addition to MSCs, class diagrams are used to provide static information used in MSCs. Therefore, more complete test suites can be specified.

Sequence diagrams are detailed by MSC which are partly introduced into sequence diagrams of UML 2.0. MSC provides, for instance, description of message- and procedure-based communication, usage of timer, in-

line expressions to describe, for instance, alternatives, and HMSCs to describe sequences of MSC execution. The new data and time concept to describe usage of an external data language and time dependencies was not detailed because it is not introduced into UML until now.

The IDL was introduced which permits the specification of software interfaces. The interfaces are independent of the programming language or the platform which is used and can be automatically generated from UML class diagrams.

If TTCN-3 is used for testing systems with interfaces specified by IDL, these interface definitions can be used as ATSI. Therefore, the mapping suggestion which will be given in chapter 5 can be used to generate the *static* ATS parts automatically. This would effect definitions like data types and signatures for procedures. Hence, interface modifications could be seamlessly introduced into the *static* part of TTCN-3 test suites which would improve consistence and allow simplified test specification via UML models or testing of CORBA-based systems.

Outlook

The progress of UML to UML 2.0 will enhance support for automatic test generation because of more semantics information and more expressive diagrams like sequence diagrams which are enhanced by MSC features and state diagrams wich are enhanced by SDL features. For instance, tool support for UML 2.0 is given by second generation of Telelogic TAU tools.

Furthermore, in conjunction with the new approach *Model Driven Architecture* (MDA) (OMG 2001a) the UML and XML *Metadata Interchange* (XMI) is used to specify systems where full automatic code generation, simulation, and validation gets possible. It is thought to use them for describing standards, too (Koch 2001). MDA separates system specification from implementation specification on a specific technology platform by specifying, for instance, architectures for models and a set of guidelines to structure specification by models.

To use MDA for specific application areas UML profiles may be defined. Thus, UTP gets introduced which will widen usage of UML for testing. Especially, in conjunction with the mapping of UTP to TTCN-3 as it will be defined in ITU-T Recommendation Z.149. A case study can be seen in Dai et al. (2004). Automatic code generation using MDA increases the demand for conformance testing, certification, and branding. Using TTCN-3 in conjunction with *eXtensible Markup Language* (XML) is shown, for in-

stance, in Schieferdecker & Stepien (2003). Additionally, usage of IDL for different application areas is detailed in section 5.7.

In particular, automatic test generation and full test specification can be simplified through better semantics, more expressive features, and combination with TTCN-3. The approach described in this thesis is a first step in this new direction but focusses more on test case specification instead of a full test suite.

4. Mapping of MSC to TTCN-3

CATG based on state space exploration, as described in section 2.3 on page 14, generates frequently inefficient test cases. Therefore, it is desirable to use, for instance, graphical test purposes, for CATG which are also more suitable for a test designer. Test case generation and specification using MSCs were formerly done for TTCN-2 by using MSC-96 and is provided, for instance, with AUTOLINK in the TAU tool set from TELELOGIC (see section 2.3 on page 15). Hence, a mapping from MSC to TTCN-3 (see section 2.4) gets defined to allow definition of graphical test purposes for TTCN-3, too (Ebner 2004).

The focus is on timed order of message exchanges and test suite details are hidden. This distinguishes this concept in contrary to UTP where a test suite is represented in more detail. Thus, specification of scenario-based test cases gets simplified. Furthermore, usage of a given specification by MSCs or UML diagrams which can be converted into MSCs is possible. As stated in section 3.4 the MSC-2000 is used as substitute for UML sequence charts. UML activity diagrams can be converted to HMSC which is used to define test execution order (see Figure 4.1). Thus, MSC is used to generate TTCN-3 test cases and control parts.

MSC is also used for real-time testing and MSC concepts have been used to develop GFT and UTP. The introduction of new concepts for real-time testing with TTCN-3 and MSC is discussed in Dai et al. (2002, 2003); Neukirchen (2004). The GFT in context of MSC and UML is discussed in Schieferdecker & Grabowski (2003). The deployment of UTP is detailed in Schieferdecker et al. (2003) where a UML-based specification of test descriptions is explained.

The mapping starts with a description of the used concept to motivate the following parts which are structured similar to the MSC specification document (ITU-T 2001) to provide easy access to the mapping of each MSC element. Therefore, mapping of the MSC documents and comments, basic MSCs, structural concepts, and HMSCs are explained. Finally, a summary and an outlook are given.

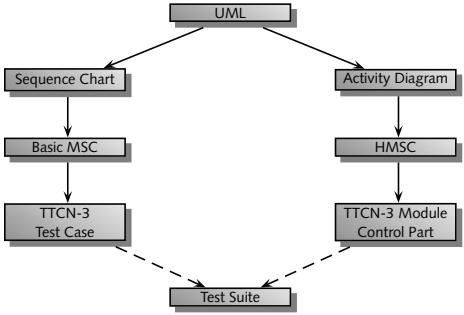


Figure 4.1.: *Basic* MSC *to* TTCN-3 *mapping concept*

4.1. Fundamental Concept

Contrary to the TTCN-3 presentation format GFT (ETSI 2002b) and UML test specification via UTP (Schieferdecker et al. 2003) which are inspired by MSC the concept here uses MSC to manually specify test purposes and cases. However, the concept is also not using the full MSC semantics because some restrictions and adaptations respectively have to be done to permit test case specification via MSC. Thus, MSC semantics is overwritten by an own MSC to TTCN-3 semantics which is as near as possible to the MSC semantics without introducing new graphical elements. Hence, seamless use by test designer familiar with MSC is supported.

Requirements

There are some requirements which are desirable to provide good support for TTCN-3 with MSC and to widen usability and acceptance:

Structure It should be feasible to control the TTCN-3 test case generation process to get a desired test suite structure. For instance, the MSC document structure can be used.

Aliases Use of aliases is important to write easily test cases via MSCs wherefore usage of TTCN-3 templates has to be supported.

Statements Support insertion of TTCN-3 statements (program code) in MSC, for instance, insertion of **setverdict(pass)** at the end of an MSC diagram.

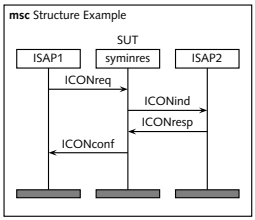


Figure 4.2.: *Basic instance structure to specify test cases via* MSC

Defaults Usage of predefined TTCN-3 defaults must be possible.

Concurrency Support concurrent and non-concurrent TTCN-3 wherefore non-concurrent should be a special case of concurrent TTCN-3.

Basic Structure

In order to use MSC for generating TTCN-3 the test architecture and communication has to be mapped first (see subsection 2.4.4 on page 29 and subsection 2.4.5 on page 31). Static information like types and data has to be given by other sources as shown in chapter 5. Test architecture and communication is based on components and ports whereby ports are the element which connects both together. Furthermore, usage of components would be too abstract for scenario-based testing. Therefore, MSC instances are mapped to ports. At least the static MTC and SUT ports have to be used to specify test cases (see Figure 4.2). Dynamic PTCs can be represented by instance creation. In order to distinguish ports by components the instance head provides the component name in addition. Hereby, an own test case for each component can be generated in case of concurrent TTCN. In case of non-concurrent TTCN usage of PTCs is forbidden and component name MTC is optional and only test case(s) for the MTC are generated.

In case where an MSC is ambiguous in sense of a TTCN-3 test case several test cases will be generated to comply the ambiguities. Nevertheless, an ambiguous test case specification in sense of this concept cannot be seen as a test purpose in its proper meaning because traces have not to be completed. Ambiguous test cases can appear by using sending messages in front of alternative inline expressions and must appear by using HMSCs.

Restrictions and Adaptations

As stated before, some restrictions and adaptations are made to use MSC in conjunction with TTCN-3:

Configuration file A configuration file has to provide the used *data types*, *templates* (constraints), and configuration information like definition of *ports* and their types and components (for create, connect, visibility) which can be provided by a SDL or IDL specification, for instance.

The configuration file has to be a correct TTCN-3 module. A reference to the configuration file can be given via a text comment in the MSC because the generated test case will be inside a TTCN-3 module, too.

Synchronisation At the border of inline expressions synchronisation is always assumed to prevent cases of *interleave* and to permit loop abortion in loop expressions. MSC references are synchronised per component.

Synchronise conditions and local synchronise rules will be used and hence, no global synchronisation and global references are available.

Message type In an inline expression the first message has to be type limited to make sense in TTCN-3. Therefore, only *receive messages* and *receive statements* respectively are permitted as first message in an alternative, optional, and exceptional inline expression.

4.2. MSC Documents and Comments

MSC documents are used to structure test cases in a TTCN-3 module wherefore the MSC document name is used as module name and the referenced MSCs are converted to appropriate test cases inside the module. The document *relation* is used to bind a configuration file to the MSCs where static information are provided.

The three comment types *note*, *comment*, and *text* are used to insert TTCN-3 statements and comments. Notes occur only in the textual syntax and therefore, may be used in comment types *comment* or *text* to insert TTCN-3 comments. Comments are associated with a symbol or comment type *text* why they can be used to insert TTCN-3 statements at

special positions. *Text* is used for global TTCN-3 comments and statements which have to be set at the beginning of the generated TTCN test case. For instance, defaults can be activated in *text* comments.

4.3. Basic Message Sequence Charts

The conversion concepts for basic MSC are given below. Instances, messages, and control flow is discussed first and the ordering and synchronisation of events gets explained afterwards. At the end the conversion of environment and gates, actions, timers, and dynamic instances is detailed.

Each chart contains a chart name which gets the test case name which may include used parameters.

4.3.1. Instances, Messages, and Control Flow

Instances and communication by message exchange and procedure invocation are the basic parts of MSCs. According to the concept mentioned before instances are used as ports and PCO respectively where the corrsponding component is mentioned, too. Instance *name* is used as port name and instance *kind* without denominator is used as corresponding component name (see Figure 4.3).

The instance definition itself provides no further information but attached elements are considered to belong to the port or component respectively. Hence, attached messages and procedure invocations belong to the given port and all others to the component why all elements are put into the same component test case(s). The used port type like **message**, **procedure**, or **mixed** may be given in the instance name but is only necessary if semantic checks will be done without further available configuration information or because of consistency for test designer. The concrete port definitions are given by the separate static information in the configuration file as also done for the component definition.

If no architecture information beside ports and SUT is given a heuristic can be used or only a MTC is assumed. It is indicated which ports belong to the SUT. Message interchange, where no SUT instance is involved, indicates use of a PTC as far as the MTC sends no message to itself. For instance, in case of concurrent TTCN-3 synchronisation between components is done via synchronisation coordination messages. Furthermore, test cases are focussed on communication with the SUT why SUT instances are mostly involved in message interchange. However, usage of such heuristics is up to tool vendors because they have no influence to the mapping itself.

An MSC message and procedure consists of a relation between an output and input event from and to an environment or instance. These can be matched to send and receive operations in TTCN-3. Output message events can only be matched to **send** and output procedure events to **call**, **reply**, or **raise** operations whereas input message events can be matched to **receive**, **trigger**, and **check** and input procedure events to **getcall**, **getreply**, and **catch** operations. It is possible to provide a TTCN-3 statement attached to an event by a *comment* to force its conversion to a specific communication operation. The default conversion converts to **send** and **receive** for messages and **call** and **reply** for procedures if a statement is missing. Value assignments and optional parameters can also be provided in comments to a communication event. One-to-many connections are not supported until now by MSC and therefore, are not considered here.

Procedure calls in TTCN-3 have a blocking characteristic to state whether test case execution is blocked or not until the call has returned by a response or exception. Thus, MSC asynchronous and synchronising calls are mapped to TTCN-3 non-blocking and blocking calls.

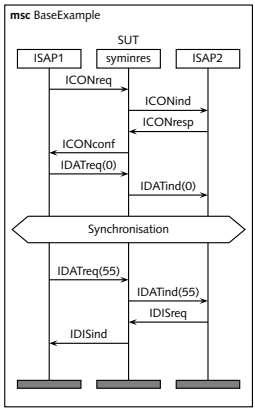
The message and procedure names represent either the used type (message type or procedure) or template (message or signature template). In case of a template without parameters no parameter list is given. Otherwise, the parameter list is used to provide the necessary parameters. TTCN-3 templates and their arguments provide easier usage of communciation operations because there are no further information necessary. Inline templates for matching receiving events can be used, too. Thus, test cases get more understandable and maintainable.

4.3.2. Ordering and Synchronisation

To specify and implement test case generation from MSC the specification of the underlying event ordering is fundamental. Using MSC event ordering implies generation of test cases for each possible trace through a chart. However, intention of using sequence diagrams by test designer directly is control of generated test cases. Thus, a limited MSC event ordering gets used. In the following only non-concurrent TTCN-3 is mentioned to keep explanation simpler. Concurrent TTCN-3 differs mainly in generation of test cases for each component which can be easily adopted by considering involved instances only (Grabowski et al. 1999).

Event ordering in MSC is defined by

- total ordering on instance axis,



(a) MSC

```
testcase BaseExample() runs on syminres {
    ISAP1.send( ICONreq )
    ISAP2.receive( ICONind )
    ISAP2.send( ICONresp )
    ISAP1.receive( ICONconf )
    ISAP1.send( IDATreq ( 0 ) )
    ISAP2.receive( IDATind ( 0 ) )
    ISAP1.send( IDATreq ( 55 ) )
    ISAP2.receive( IDATind ( 55 ) )
    ISAP2.send( IDISreq )
    ISAP1.receive( IDISind )
}
```

(b) TTCN-3

Figure 4.3.: MSC *to* TTCN-3 *base mapping*

- partial ordering between instances,
- a general ordering mechanism, and
- coregions.

Total ordering on instance axis is also used because event order on a port has to be ordered totally by default. If another behaviour is wished the ordering has to be modified explicitly by using corresponding elements. Partial ordering between instances can enable several permitted traces of a chart which prevents exact test case design because of unwished traces. Thus, partial ordering is limited by *synchronisation points*, send events are executed as early as possible, and graphical order is used to decide about used trace if necessary. Graphical order means ordering is given by the order from left to right and top to bottom in the MSC until the end or a synchronisation point.

Synchronisation points are used to provide a well defined point where all send and receive events have been executed. Event execution before is forced and no event after can be moved before a synchronisation point. Due to the limitation of the partial ordering between instances and the preference of the graphical order the general ordering mechanism is not necessary but can be used for additional, more precise, and explicit order

information. Coregions are used to override explicitly any ordering by all permutations of the involved events which is done by the **interleave** statement in TTCN-3. If the above mentioned rules for ordering can be widen by automatic detection of *interleave* cases like reception of two consecutive events at the SUT an **interleave** statement shall be used for it.

Synchronisation can be forced by using conditions (see Figure 4.3). It is assumed around all inline expressions and actions. MSC references are synchronised per component. Since synchronise conditions and local synchronise rules like described before are used, no global synchronisation and global references are available.

4.3.3. Miscellaneous

Mapping of environment, actions, conditions, timers, and dynamic instances gets described in the following.

Environment and Gates

Support of external message exchange can be designed similar to the normal message exchange by handling the environment like another instance. Gates require no special mapping rules because they are only used for better organisation of charts.

Actions

An action describes an internal activity of an instance and hence, it can be used to insert comments and TTCN-3 statements directly into test cases depending on an instance.

Conditions

Conditions are only used for synchronisation which requires coordination messages if concurrent TTCN-3 is used.

Timers and Time Constraints

Timers can be started and stopped according their position in the MSC and global timers are considered. Timer time out can be catched by using the default behaviour statement of TTCN-3 where the test case verdict can

be set to **fail** and a log message can be written. There is no stop after the verdict to allow ending in a defined state.

Instance Creation and Termination

Dynamic instances are mapped like static instances but are created and started only during test case execution and not at the begin of a test case like it is done for MTC and SUT. However, dynamic instances make only sense if a new component is used. Information for connecting and mapping of ports to each other can be taken from exchanged messaged. Instance termination is mapped to the **stop** component operation but can only be inserted if no other port of the component is in use.

4.4. Structural Concepts

Conversion of coregions, references, instance decomposition, and inline expressions, which are all summarised as structural concepts in MSC, are detailed now.

4.4.1. Coregions

Coregions are used to override explicitly the total ordering of an instance axis by unordered events. In sense of a test case description best mapping is done by using all permutations of the involved events which is done by the **interleave** statement in TTCN-3, as well. This was mentioned earlier in subsection 4.3.2.

Usage of coregions is limited on message exchange among components and between SUT and a component. Overlapping of coregions on different instance axes for the same component has to be prevented. The first event must be always a receive event. In case of non-concurrent TTCN-3 coregions are only used on SUT axes.

4.4.2. MSC References

MSC references provide usage to import other charts which easies decomposition and reuse. However, test cases cannot be decomposed into other test cases and therefore, references are converted to function calls. Nevertheless, test case decomposition is done by HMSCs which is described in section 4.5 on page 75. To allow correct behaviour of function calls by

references they have to be synchronised per test component. Several function calls can be provided in one reference by usage of the **seq** operator for references.

The *sequential* MSC reference expression **seq** is used to call several functions in sequential order. The *alternative* and *optional* expression **alt** and **opt** is used to generate a test case for each possible alternative. *Loop* expressions are converted according done for MSC inline expressions in HMSCs (see section 4.5).

4.4.3. Instance Decomposition

Instance decomposition is used to replace an instance axis by a detailed chart. The instance axis can be seen as the environment from the detailed chart. Therefore, instance decomposition has only to be resolved during conversion wherefore no further special conversion concept is required.

4.4.4. Inline Expressions

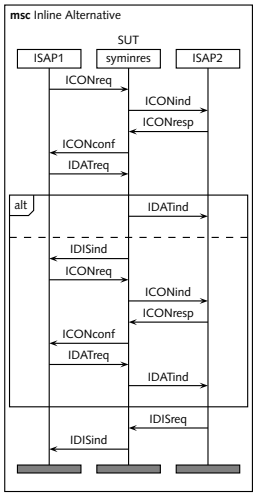
In order to structure charts inline expressions are provided which allow alternatives, loops, and parallel execution.

An appropriate mapping requires synchronisation before and after each inline expression to prevent cases of *interleave* which cannot be solved easily. In addition, handling concurrent TTCN-3 gets better. From MSC point of view events before and after an inline expression may influence the expression behaviour but for test case design this behaviour is not wished because it makes test case design more complicated by loosing control about the number of generated test cases. To provide a good mapping to TTCN-3 alternative and loop statements synchronisation is required, as well. Especially, good support of *loop* expressions by using TTCN-3 loop statements is also wished.

The inline expressions *alternative*, *option*, *exception*, and *loop* are detailed below. Inline expression *parallel* will not be mentioned because there is no appropriate mapping available.

Alternative

The *alternative* expression is directly mapped to the alternative behaviour statement of TTCN-3. Therefore, the first element of each alternative part has to be a reception statement according the specification of the **alt** statement (see subsection 2.4.6 on page 34). For each alternative of an



(a) MSC

```
testcase  Inline_Alternative ()  runs on syminres {
    ISAP1.send( ICONreq )
    ISAP2.receive( ICONind )
    ISAP2.send( ICONresp )
    ISAP1.receive( ICONconf )
    ISAP2.send( IDISreq )
    ISAP1.send( IDATreq )
    alt  {
        []  ISAP2.receive( IDATind )
            {
            }

        []  ISAP1.receive( IDISind )
            {
                ISAP1.send( ICONreq )
                ISAP2.receive( ICONind )
                ISAP2.send( ICONresp )
                ISAP1.receive( ICONconf )
                ISAP1.send( IDATreq )
                ISAP2.receive( IDATind )
            }

    }

    ISAP2.send( IDISreq )
    ISAP1.receive( IDISind )
}
```

(b) TTCN-3

Figure 4.4.: MSC *to* TTCN-3 *alternative mapping*

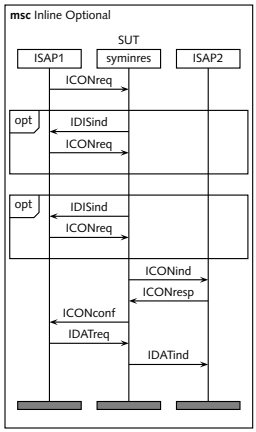
alternative expression containing a send event as first event another test case containing this alternative is generated. In worst case an own test case gets generated for each alternative. Of course, all events in alternatives are restricted by the **alt** statement semantic (see Figure 4.4).

Optional

The *optional* expression can be seen as a special case of the alternative expression with an additional *empty* alternative part. The empty alternative part gets realised by an **else** guarded empty alternative part (see Figure 4.5).

Exception

The *exception* expression is a special case of the *alternative* expression where the last alternative part is the remainder of the MSC (see Figure 4.6).



(a) MSC

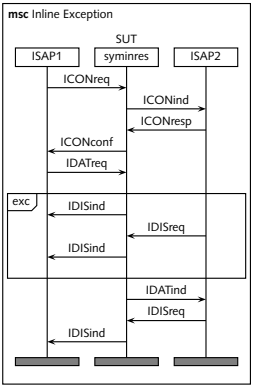
```
testcase Inline_Optional () runs on syminres {
    ISAP1.send( ICONreq )
    alt {
        [] ISAP1.receive( IDISind )
           {
               ISAP1.send( ICONreq )
           }

        [else]
               {
               }

    }

    alt {
        [] ISAP1.receive( IDISind )
           {
               ISAP1.send( ICONreq )
           }

        [else]
               {
               }

    }

    ISAP2.receive( ICONind )
    ISAP2.send( ICONresp )
    ISAP1.receive( ICONconf )
    ISAP1.send( IDATreq )
    ISAP2.receive( IDATind )

}
```

(b) TTCN-3

Figure 4.5.: MSC *to* TTCN-3 *optional mapping*

Loop

The *loop* expression can be converted onto the **for** and **while** loop statements of TTCN-3 whereby infinite loops are best mapped to **while** loops (see Figure 4.7) and finite loops to **for** loops (see Figure 4.8). In contrary to the alternative expression there is no special first event necessary. Instead, loop operations require an abortion criteria if lower and upper boundary are *not* equal. Equal boundaries impose exact number of loop passes where no abortion criteria is required. The abortion criteria from MSC point of view is the next receive event after the loop expression. Therefore, a receive message has to be given after each loop expression. The receive message gets checked inside the loop and concrete message consumption is done after loop execution.

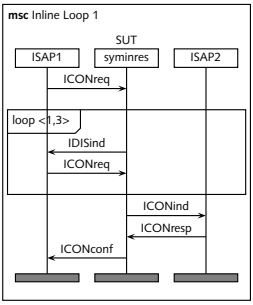


(a) MSC

```
testcase Inline_Exception () runs on syminres {
    ISAP1.send( ICONreq )
    ISAP2.receive( ICONind )
    ISAP2.send( ICONresp )
    ISAP1.receive( ICONconf )
    ISAP1.send( IDATreq )
    alt {
        [] ISAP1.receive( IDISind )
           {
               ISAP2.send( IDISreq )
               ISAP1.receive( IDISind )
           }

        [else]
               {
                   ISAP2.receive( IDATind )
                   ISAP2.send( IDISreq )
                   ISAP1.receive( IDISind )
               }
    }
}
```

(b) TTCN-3

Figure 4.6.: MSC *to* TTCN-3 *exception mapping*

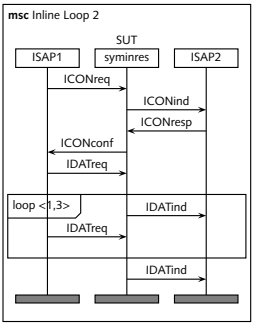


(a) MSC

```
testcase Inline_Loop1() runs on syminres {
    ISAP2.send( ICONresp )
    ISAP1.send( ICONreq )
    while ( true ) {
        alt {
            [] ISAP1.receive( IDISind )
               {
                   ISAP1.send( ICONreq )
               }

            [ iterator__1 >= 0] ISAP2.receive( ICONind )
                                {
                                    goto iterator__1_label
                                }

        }

    iterator__1 := iterator__1 + 1;
    }

    label iterator__1_label ;

    ISAP1.receive( ICONconf )
}
```

(b) TTCN-3

Figure 4.7.: MSC *to* TTCN-3 *loop mapping 1*



(a) MSC

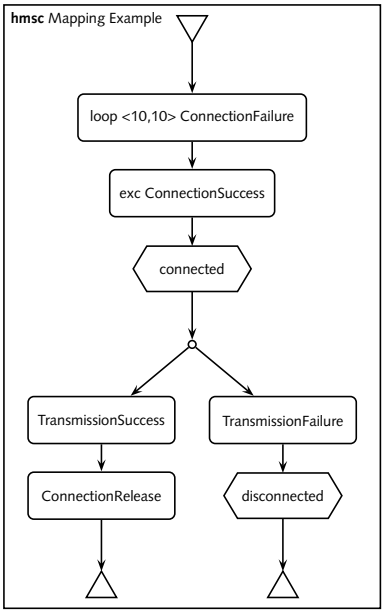
```
testcase Inline_Loop2() runs on syminres {
    ISAP1.send( ICONreq )
    ISAP2.receive( ICONind )
    ISAP2.send( ICONresp )
    ISAP1.receive( ICONconf )
    ISAP1.send( IDATreq )
    var integer iterator__1 := 0;
    for ( iterator__1 := 0; iterator__1 < 3;
          iterator__1 := iterator__1 + 1 ) {
        alt {
            [] ISAP2.receive( IDATind )
               {
                   ISAP1.send( IDATreq )
               }

            [ iterator__1 >= 1] ISAP2.receive( IDATind )
                                {
                                    goto iterator__1_label
                                }
        }
    }
    label iterator__1_label ;
}
```

(b) TTCN-3

Figure 4.8.: MSC *to* TTCN-3 *loop mapping 2*

4.5. High-Level Message Sequence Charts

HMSCs are thought to describe possible combinations of MSCs. As mentioned in section 3.3, UML activity diagrams are comparable with HMSCs. If they are seen as a kind of test suite decomposition they can be used to specify test case execution order. Therefore, HMSCs are used to describe test case execution in the module control part of TTCN-3.



(a) HMSC

```
module {

  function testExecution1 () {
    verdicttype verdict;
    integer i;

    for (i:=10; i>0; i:=i-1)
      execute ConnectionFailure();

    verdict = execute ConnectionSuccess();
    if (verdict == false)
      return;

    execute TransmissionSuccess();
    execute ConnectionRelease();
  }

  function testExecution2 () {
    verdicttype verdict;
    integer i;

    for (i:=10; i>0; i:=i-1)
      execute ConnectionFailure();

    verdict = execute ConnectionSuccess();
    if (verdict == false)
      return;

    execute TransmissionFailure ();
  }

  control {
    testExecution1 ();
    testExecution2 ();
  }
}
```

(b) TTCN-3

Figure 4.9.: MSC *to* TTCN-3 *mapping of* HMSCs

MSC references including reference expression in HMSCs are converted to corresponding test case execution calls and conditions and parallel frames are ignored. Each execution trace of a HMSC gets collected in an own function and all functions are called in the control part. No backward loops are allowed and there are only reference expression loops with same finite lower and upper boundary allowed because data concept is not introduced into UML and thus, not considered here. Alternative and

optional expressions lead to several traces only. Exception expressions are used to stop further trace execution if the test case fails. An example is given in Figure 4.9.

4.6. Summary and Outlook

Summary

The chapter describes a mapping of MSC elements to TTCN-3 statements in order to use UML sequence charts to define test cases and test case execution order in TTCN-3.

A conceptual mapping list is given in Table 4.1. Usage of comments for introducing direct TTCN-3 statements is used especially for TTCN-3 *defaults*. One-to-many connections are not supported until now by MSC and therefore, are not considered. Apart from mapping chart elements the ordering semantics of events had to be defined which differs from the original semantics of MSC to be adequate for test specification. For instance, default synchronisation points were introduced for references and inline expressions.

Procedural communication is represented by flow control concept and mapped to **call** and **getreply** statements. However, alternative handling of reception events by usage of alternative statement in a blocked call or a **getreply** statement is not supported until now. Data and time concepts of MSC-2000 have not been considered because they are not used in UML 2.0, until now. Nevertheless, in context of *Timed*TTCN-3 it has been shown that is possible to translate real-time information contained in MSC to *Timed*TTCN-3 (Dai et al. 2002, 2003; Grabowski 2002; Neukirchen 2004). Data support was not considered but can be easily used to enhance expressiveness and integration into TTCN-3.

A prototype was implemented to demonstrate feasibility. A first version was demonstrated on the TTCN-3 launching event in October 2000 at *European Telecommunications Standards Institute* (ETSI). Most examples in this chapter are converted with it. The prototype is based on the MSC parser developed at the Institute for Telematics, University Luebeck, Germany. The parser was implemented via *ANother Tool for Language Recognition* (ANTLR) which "is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, or C++ actions" (http://www.antlr.org). The MSC parser is used to build an *Abstract Syntax Tree* (AST) first which will be given to a tree parser. The tree parser parses the AST and generates

Table 4.1.: *Conceptual list of* MSC *to* TTCN-3 *mapping*

MSC	*TTCN-3*
basic MSC	test case
HMSC	module control part
instance axis	represents a port
note	comment
comment	insert statements directly at element position
text comment	insert statements directly at beginning
chart name	test case name
instance name	port name
instance kind	component name or SUT
message	send or receive statement
flow control	call or getreply statement
action	insert statements directly
condition	marks a synchronisation point
timer	time start, stop, or timeout statement
instance creation	create componenent
coregion	interleave statement
reference	function call or test case execution call (depending on MSC or HMSC)
alt,exc, or opt expression	alt statement or function calls (depending on MSC or HMSC)
loop expression	for or while statement
par expression	not used

TTCN-3 code by using C++ inline code. The prototype shares code with the IDL to TTCN-3 converter prototype mentioned in chapter 5.

Outlook

In further work the prototype should support concurrent TTCN-3 and HMSC which have been explained but not implemented until now. A discussion of concurrent TTCN-2 and MSC can be found in Koch (2001). Usage of TTCN-3 as data language has also to be considered. There is a prototype enhancement available which supports the *time* concept as described by *TIMED*TTCN-3 (Dai et al. 2003; Neukirchen 2004). Industrial interest on the prototype has been shown.

5. Mapping of IDL to TTCN-3

In conjunction with UML-based manual test purpose specification, as described in chapter 4, the usage of static information like types and interfaces for test specification was mentioned in section 3.3. The IDL (see section 3.5) was selected as intermediate format to provide a transition from UML class diagrams to TTCN-3 (see section 2.4). IDL is well supported by UML tools and widens applicability because of the wide usage of IDL or similarities with IDL. For instance, IDL is used in CORBA, is similar with *Simple Object Definition Language* (SODL), and a mapping between WSDL and IDL exists.

Usage of IDL for TTCN-3 is described by a mapping of IDL elements to corresponding TTCN-3 elements. Each mapping consists of rules to map an IDL element and rules which have to be considered in the IDL definition to prevent conflicts with TTCN-3 rules. Furthermore, there could be suggestions how to use a provided feature of IDL in TTCN-3. For example, the possibility of raising an exception and the exception type definition is defined in IDL but not catching of exceptions itself.

The mapping described here is based on some recent work in Ebner (2001a, b); Ebner et al. (2002); Yin (2001); Yin et al. (2001). and is also summarised as ETSI Technical Specification 102 219 (ETSI 2003).

Firstly, a description of the used approach gets described. Secondly, the mapping of the lexical conventions and preprocessing is explained. Afterwards, the mapping of the IDL structuring elements and types is detailed. Fifthly, communication declaration mapping is shown. Sixthly, mapping of names and scoping are described. Finally, a summary and an outlook are given.

5.1. Fundamental Concept

Two different approaches can be followed: either using the *implicit* or the *explicit* mapping. The *implicit* mapping makes use of the *import* mechanism of TTCN-3, denoted by the keywords **language** and **import**. It facilitates the immediate use of data specified in other languages. Therefore, the definition of a specific data interface for each of these languages is required. Currently, ASN.1 data can be used besides the native TTCN-3

types. The data interface for IDL types and values is still a topic of ongoing research. The work presented here follows the approach of *explicit* mapping, that is IDL data are converted into appropriate TTCN-3 data. Only those TTCN-3 data are used further in the test specification.

There are former mappings for TTCN-2, but TTCN-3 provides new features as stated before (Li 1998; Mednonogov 2000; Mednonogov et al. 2000; Schieferdecker et al. 1998). The old mapping of IDL to TTCN-2 has to be redesigned to make use of these new features. It was intended to describe a mapping which is independent of implementation details wherewith it can be used in general. For instance, in conjunction with CORBA it is assumed that a CORBA/TTCN-3 gateway executes the concrete mapping between the TTCN-3 test suite and the CORBA-based SUT. Furthermore, it was not intended to map IDL to ASN.1 data as it was used in Mednonogov (2000); Mednonogov et al. (2000).

5.2. Lexical Conventions and Preprocessing

The *lexical conventions* of IDL define *comment*, *identifier*, *keyword*, and *literal* conventions which are described below. Thereafter, some comments to the preprocessing support of IDL are given.

Comments

Comments can be defined in IDL and TTCN-3 by using the pair "**/*** " and "***/**" for comment blocks or "**//**" for end of line comments like defined in ISO C++ (ISO/IEC 1998c).

Identifiers

Identifiers in IDL and TTCN-3 consist of alphabetic, digit, and underscore characters where the first character must be an alphabetic character and all characters are significant. However, there is no case distinction in IDL but the spelling has to be consistent throughout the whole definition. TTCN-3 uses case sensitive identifiers and therefore, the IDL rule defines a subset of the TTCN-3 rule.

Keywords

It has to be made sure that no TTCN-3 keywords (see ETSI 2002a, appendix A.1.5) are used as identifiers in the IDL definition if a seamless

Table 5.1.: IDL *to* TTCN-3 *literal mapping*

Literal	*IDL Convention*	*TTCN-3 Convention*
Integer	no "0" as first digit	no "0" as first digit
Octet	"0" as first digit	'FF96'O[1]
Hex	"0X" or "0x" as first digits	'AB01D'H
Floating	1222.44E5 (Base 10)	1222.44E5 (Base 10)
Char	't'	"t"
Boolean	TRUE, FALSE	true, false
String	"text"	"text"
Wide string	"text"	"text"
Fixed point	12.34D	12.34

identifier mapping is wished. However, identifiers can be renamed, for example, by appending a special prefix or suffix in case of a conflict with keywords in TTCN-3. Concrete identifier mapping is left to tool vendors.

Literals

The definition of literals differ slightly between IDL and TTCN-3 why some changes have to be made. In Table 5.1 a mapping for each literal type is given. IDL uses the *ISO Latin-1 character set* for **string** and **wide string** literals and TTCN-3 uses ISO/IEC 646 for **string** literals and ISO/IEC 10646 for **wide string** literals (ISO/IEC 1990, 1993, 1998a).

Preprocessing

IDL preprocessing is defined by the ISO C++ standard (ISO/IEC 1998c). TTCN-3 does not support preprocessing wherefore only the **include** statement could be supported directly. The **include** preprocessor statement of IDL could be mapped onto the **import** statement of TTCN-3 to use definitions from other files. Furthermore, users could use IDL definitions by importing it and using the language option as shown here

TTCN-3

```
import all from IDLSpecification language "IDL";
```

[1] Octet literals require an even number of hexadecimal digits as given by the octetstring definition.

However, this is not always possible because **import** allows only import of types if the *module* is written in another language as TTCN-3 (ETSI 2002a, section 7.5.10). This would prevent, for instance, use of constants. Nevertheless, because of providing only an explicit mapping the import mechanism is not supported. Thus, it has to be rolled out.

All other preprocessor statements, which are mostly text replacements, are not mapped to TTCN-3 because the IDL specification should be used after preprocessing it.

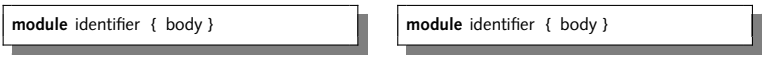
5.3. Structural Elements

IDL specifications consist of *type*, *constant*, *exception*, *interface*, *module*, and *value* declarations. Beside *basic* and *constructed* data types IDL provides the object-oriented types **interface** and **valuetype**. This object-oriented types cannot be mapped straight forward because they contain structural information which have to be considered in the TTCN-3 configuration architecture.

The mappings of modules, interfaces, values, and constants are described now. The data type and exception mappings as well as the bodies of interfaces are described later.

5.3.1. Module Declaration

IDL uses *modules* as main grouping and scoping unit. TTCN-2 mappings use modules to define nested *test groups* whereas TTCN-3 owns an own module concept. For this, the module concept of TTCN-3 is used.



5.3.2. Interface Declaration

Interfaces describe objects with all their access methods by using *operations* and *attributes*. Additionally, interfaces can contain local type definitions like *exceptions* and *constants* which can be used by its operations and attributes. A mapping for interfaces should provide a similar scoping and grouping mechanism as well as an appropriate handling under TTCN-3 as in IDL. Since lacking an object model in TTCN-3, the group construct is used to retain the scoping information and interfaces have

to be flattened. Hence, import of single **interface** definitions from other modules via the *importing group* statement gets possible. Using **module** would not be appropriate because the module concept from IDL has to be considered as well as the module concept of TTCN-3.

The test configuration concept of TTCN-3 requires use of **component** and **port** respectively PCO. According to the TTCN-2 mappings, an interface is best mapped to a PCO. Hence, they are mapped to ports in TTCN-3. They are associated with signatures converted from attributes and operations of the interface (see subsection 5.5.1). Hereby, components are used as a collection of interfaces respectively objects.

IDL

```
interface NamingContext { body }
```

TTCN-3

```
group NamingContextInterface {
    type port NamingContext
        procedure { ... }
}

address NamingContextType ?????
```

For the specification of object interfaces IDL type **Object** is used. The interface types for application objects inherit from **Object**. Interface definitions contain structural information which have to be considered in the TTCN-3 configuration architecture. Furthermore, object references associated with instances of interface implementations can be passed over operation invocations. Since the port reference cannot be used as operation parameter or inside **signature** definitions respectively, another mechanism has to be used to represent the interface reference. TTCN-2 mappings use stringified *Interoperable Object References* (IORs) or integers. TTCN-3 does not provide special support for it but there is an **address** type available which is used to address entities in the SUT. Therefore, **Object** is generally mapped to **address** in TTCN-3 and each interface gets an **address** type declared in the data part.

Interfaces can make use of *inheritance* from other interfaces which can also be **abstract**. TTCN-3 provides no object-oriented concepts which provide inheritance wherefore all inherited elements have to be rolled out. Thus, they have to be handled as defined in the interface itself. In case of multiple inheritance elements have to be inherited only once.

Forward references of interfaces respectively ports can also be used in TTCN-3. Local interfaces require no further special mapping wherefore they can be treated as normal interfaces.

One interface can be instantiated many times but it can only be executed

properly if the corresponding port is connected to a component. Therefore, whenever a component gets created all corresponding ports respectively interfaces respectively objects are instantiated, too. Furthermore, it is not possible to dynamically change the number of objects for a component wherefore a new component has to be created. The instantiation location depends on the component which has to be matched properly by the test system. A further discussion of this problem can be found in the TTCN-2 mappings (Mednonogov 2000; Mednonogov et al. 2000).

It does not make a difference for the mapping if requested[1] or provided[2] interfaces are required by the test system and SUT. Hence, TTCN-3 can be used on client and server side without modifications to the mapping rules.

The concrete mapping of interface *operations*, *attributes*, and *exceptions* are described in section 5.5. A summary of interface mapping inclusive operation, attribute, and exception mapping can be found in Table 5.7 on page 102. General *type* mapping gets detailed in section 5.4 on page 85 and *constant* mappings are described in subsection 5.3.4 on page 85.

5.3.3. Value Declaration

The type **valuetype** is local like a **struct** but contains also *operations* and *attributes* and provides *inheritance* like by **interface** (OMG 2001b, chapter 5). If **valuetype** is used as parameter it will be passed as *object-by-value* wherefore it can be seen as an *interface* which is passed by value. In contrary to type **interface**, the IDL type **valuetype** has local operations that are not used outside the object, and are therefore not relevant from the functional testing point of view. However, since the public attributes of **valuetype** instances are used to communicate object states, we suggest to map the IDL **valuetype** types to **record** types in TTCN-3. Inheritance is treated by rolling it out as it is also defined for interfaces and given operations are mapped to **external** functions. If there is only one variable for **valuetype**, it can be mapped like given by the type mapping with a special attribute, the **variant** attribute, for this type as described in section 5.6 on page 102. Type **valuetype** was not mentioned in the TTCN-2 mappings.

The following example shows how to map **valuetype** and were used from (OMG 2001b, section 5.3, section 5.2.5.2).

[1] test system requires interface which is provided by SUT
[2] test systems provides interface which is required by SUT

IDL

```
valuetype StringValue string;

valuetype EmployeeRecord {
    private string name;
    private string email;
    private string SSN;


    factory init (
        in string name,
        in string SSN
    );

};
```

TTCN-3

```
type iso8859string StringValue with {
    encode "IDL:valuetype and IDL:string"}

group EmployeeRecordValuetype {
    type record EmployeeRecord {
        iso8859string name,
        iso8859string email,
        iso8859string SSN
    }

    external function EmployeeRecord_init(
        iso8859string name,
        iso8859string SSN );
}
```

It is interesting to remark that there is no language mapping defined to the non object-oriented language C and there is no intention to do so. Furthermore, support of type **valuetype** by the only existing ORB using C, namely ORBit, is not done or only for special cases. It is not intended to add better support because there is no user requirement for it.

5.3.4. Constant Declaration

Constant declarations can be easily converted to TTCN-3 by use of literal (see Table 5.1) and operator mapping for floating-point and integer values (see Table 5.2).

IDL

```
const long number = 017; // 017 == 0xF == 15
const long size  = ( ( number << 3 ) % 0x1F ) & 0123;
```

TTCN-3

```
const long number := '17'O;
const long size  := ( ( number << 3 ) mod '1F'H ) and4b '0123'O;
```

5.4. Data Types

IDL provides type declarations for *constant* (see subsection 5.3.4), *basic*, *constructor*, *template*, and *complex* data types. Beside these data types IDL provides the object-oriented types **interface** (see subsection 5.3.2)

Table 5.2.: IDL *and* TTCN-3 *operators for constant expressions*

Operator	*IDL*	*TTCN-3*
Unary floating-point		
positive	+	+
negative	−	−
Binary floating-point		
addition	+	+
subtraction	−	−
multiplication	*	*
division	/	/
Unary integer		
positive	+	+
negative	−	−
bit-complement	~	not4b

Operator	*IDL*	*TTCN-3*
Binary integer		
addition	+	+
subtraction	−	−
multiplication	*	*
division	/	/
modulo	%	mod
shift left	<<	<<
shift right	>>	>>
bitwise and	&	and4b
bitwise or	|	or4b
bitwise xor	^	xor4b

and **valuetype** (see subsection 5.3.3) which have been discussed before. This object-oriented types cannot be mapped straight forward because they contain structural information which have to be considered in the TTCN-3 configuration architecture.

TTCN-3 provides more predefined types than TTCN-2, for which a better mapping can be constructed. Nevertheless, the **fixed** type and the *constructed* types **union** and **any** have required special treatment to get mapped properly.

A construct for naming data types and defining new types by using the keyword **typedef** is provided by IDL. This can be done under TTCN-3 via the keyword **type**, too.

Type Extension

To enhance readability and to provide a clear distinction, mapped IDL data types under TTCN-3 may get the prefix **IDL**. Additionally, to get the same semantic meaning under TTCN-3 as given by IDL, the types get a **variant** attribute containing the source IDL type name. Therefore, a compiler or interpreter can detect the IDL dependency and can react accordingly like executing semantic checks. This includes encoding, too. This feature provides a flexible way of extending the TTCN-3 type system

and was among others introduced to simplify the IDL mapping. Use of the variant attribute is discussed further in section 5.6 on page 102.

Furthermore, based on the above discussion a library of *useful types* (see Table 2.2 on page 26) was introduced into TTCN-3 (ETSI 2002a, appendix E), too. For instance, the IDL float types **float** and **double** have a range limitation whereas the TTCN-3 **float** type has no such restriction. Thus, a correct mapping would be difficult but the type system extension solves this problem. The same can be done for the integer types as shown below.

TTCN-3

```
type integer IDLshort  with { variant "IDL:short  FORMAL/01—12—01 v2.6" };
type float   IDLdouble with { variant "IDL:double FORMAL/01—12—01 v2.6" };
```

The data type mapping will be shown in the following subsections. In order to focus on the important aspects of a mapping and to preserve readability of code examples the **variant** attribute will be omitted.

5.4.1. Basic Types

Mapping IDL basic data types to TTCN-3 data types is straight forward because they are similar to ASN.1 data types which are used as data type base in TTCN-3, too (ITU-T 1997b; Open Group 2000). For example, the **boolean** and **enumeration** types are identical in IDL and TTCN-3. Nevertheless, TTCN-3 provides more predefined types than TTCN-2 wherefore a better mapping can be provided. Their mapping is detailed below and summarised in Table 5.3 on page 91.

Integer and Floating-Point Types

All IDL *integer* and *floating point* types can be mapped to the TTCN-3 integer and floating point types which have no size limitations. To get a better corresponding for integers the range limitation according to the IDL specification is used to define the IDL types in TTCN-3 more closer.

TTCN-3

```
type integer IDLShort          ( —32768 .. 32767 );
type integer IDLLong           ( —2147483648 .. 2147483647 );
type integer IDLLongLong       ( —9223372036854775808 .. 9223372036854775807 );

type integer IDLUnsignedShort      ( 0 .. 65535 );
type integer IDLUnsignedLong       ( 0 .. 4294967295 );
type integer IDLUnsignedLongLong   ( 0 .. 18446744073709551615 );
```

TTCN-2 has no **float** type wherefore *floats* could only be mapped onto the ASN.1 **real** type. In TTCN-3 there is now an unlimited **float** type available where **float**, **double**, and **long double** from IDL can be mapped on. However, there is still no range limitation available wherefore the range limitations have to be kept in mind. Thus, the **variant** attribute is elementary for mapping float types.

TTCN-3

```
type float IDLfloat       with { variant "IEEE754 float" };
type float IDLdouble      with { variant "IEEE754 double" };
type float IDLlongdouble  with { variant "IEEE754 extended double" };
```

Because of this mapping the types are now available as *useful types* in TTCN-3 (ETSI 2002a, appendix E). The available integer types including their unsigned types are **byte**, **short**, **long**, and **longlong** and the float types are **IEEE754float**, **IEEE754double**, **IEEE754extfloat**, and **IEEE754extdouble** (see Table 2.2 on page 26). In the remainder the *useful types* are preferred.

Char and Wide Char Type

The IDL types **char** and **wchar** represent a single and wide character.[3] TTCN-3 introduces the character types **char** and **universal char** wherefore IDL **char** and **wchar** can now be mapped properly on it. However, TTCN-3 uses ISO/IEC 646 as character set for type **char** and ISO/IEC 10646 for type **universal char**. Therefore, the IDL specification has to be limited to ISO/IEC 646 respectively ISO/IEC 10646 or an appropriate conversion has to be used (ISO/IEC 1990, 1993).

IDL

```
const char letter     = 'A';
const wchar wideLetter = 'A';
```

TTCN-3

```
const char letter                := "A";
const universal char wideLetter := "A";
```

Boolean Type

The IDL **boolean** type can be mapped directly to the TTCN-3 **boolean** type.

IDL

```
const boolean isValid = TRUE;
```

TTCN-3

```
const boolean isValid = true;
```

[3] Any character set can be used for type **wide char**

Octet Type

Type **octet**, an 8-bit quantity, is not mapped onto an integer type because it has the special feature that it will not change its internal ordering if transfered between different system architectures. To represent it **octet** is mapped to **octetstring**.

IDL

```
const octet data = 0x55;
```

TTCN-3

```
const octetstring data = ''55''H
```

Any Type

In IDL it is possible to represent each possible IDL type with the **any** type. There was no corresponding type in TTCN-3 available but based on the following discussion an **anytype** was introduced into TTCN-3 (ETSI 2002a, section 6.4). The TTCN-3 **anytype** is a shorthand for the *union* of all *known types* in a TTCN-3 module.

One possible way is to use an **union** type to store all possible data types. The **union** type comprises all required types whereas these types have to be known in advance. In order to avoid this restriction an **union** type for all possible types in TTCN-3 used by the IDL mapping rules could be defined. However, it is only possible to use basic data types and well defined constructed types wherefore no generic constructed types for **set**, **record**, **union**, etc. are supported. For instance, it is not possible to provide entries for all possible kinds of type **set** by using a generic **set** type. Hence, the user could define his own *restricted* **any** type in TTCN-3 by using a **union** type containing only all possible types of the concrete application and not all thinkable types. This new **any** type has to be mapped to a full **any** type by the test system. However, this requires a careful handling of **any** types because some type definitions could be missing.

The mapping for TTCN-2 is bound to ASN.1 and therefore, it encounters problems in distinguishing types which are mapped onto the same ASN.1 type (Mednonogov et al. 2000). There is no support of the **any** type given but the use of the design pattern *decorator* is suggested if **any** type is used. Another mapping provides only basic types for the **any** type and leaves structured types open for further study (Li et al. 1999). The use of type **variant** as described on page 86 is not appropriate to solve the **any** type problem, neither. This is because of the TTCN-3 type handling which makes it impossible to handle types which are not known before-

hand. CORBA provides use of a *dynamic management of any values* but it is not appropriate to use this concept for a mapping of the **any** type here because it would not be a general mapping (OMG 2001b, chapter 9).

Since data types used by the test system are usually already known at compile time as it is now the case in TTCN-3, it is suggested to use an **union** type for data that are communicated to/over the SUT by the IDL **any** type. The **union** type must cover all data types, either basic or derived, that are converted from the IDL specification used by the test system. An **any** type for basic types could look like the one below where the *type kind* is stored in a separate variable.

TTCN-3 *helper data type for* IDL any *type (1)*

```
type record IDLAny {
    IDLTCKind kind,
    IDLAnyValueHelper value_ }

type union IDLAnyValueHelper {
short                 short_,
long                  long_,
longlong              longLong_,
unsignedshort         ushort_,
unsignedlong          ulong_,
unsignedlonglong      ulongLong_,

IEEE754float          float_ ,
IEEE754double         double_,
IEEE754extdouble      longdouble_,

octetstring           octet_,
boolean               boolean_,

char                  char_,
universal char        wchar_
}
```

TTCN-3 *helper data type for* IDL any *type (2)*

```
type enumerated IDLTCKind {
tk_null , tk_void,

tk_short, tk_ushort,
tk_long, tk_ulong,
tk_longlong, tk_ulonglong,

tk_float , tk_double, tk_longdouble,

tk_char, tk_wchar,
tk_string , tk_wstring,

tk_octet, tk_boolean,
tk_any, tk_objref ,

tk_struct , tk_union, tk_sequence,
tk_enum, tk_array, tk_fixed

tk_alias , tk_except, tk_TypeCode,
tk_Principal , tk_value, tk_value_box,
tk_native, tk_none, tk_abstract_interface
}
```

However, using **any** in an interface can nowadays be seen as a gap in a good system and interface specification. Thus, use of the **any** type should be prevented what is also done in ASN.1 (ITU-T 1997b, appendix E.3). Therefore, the **any** type should not occur because, if tests via TTCN-3 are executed, the interfaces should be quite stable without use of the **any** type. Nevertheless, there are scenarios where the **any** type has to be used. A quite common example is an application with a graphical user interface which forwards data without looking into it.

Table 5.3.: IDL *to* TTCN-3 *mapping for basic types*

IDL	*TTCN-2/ASN.1*	*TTCN-3*
short, long, longlong	INTEGER	integer with range limitation
float, double, long double	—	float
boolean	BOOLEAN	boolean
octet	OCTET STRING	octetstring
char	GraphicString, IA5String(SIZE(1))	char
wchar	GraphicString, BMP-String(SIZE(1))	universal char
any	CHOICE	anytype (respectively union)

The basic type mappings are summarised in Table 5.3.

5.4.2. Constructed Types

IDL provides the three *constructed* types **struct**, **union**, and **enum**. Recursive construction of types is only permitted with the **sequence** *template* type (see section 5.4.3).

There is no fundamental difference between mapping to TTCN-2 and TTCN-3 for most *constructed* types given because the new data types in TTCN-3 are directly introduced from ASN.1. Hence, only a closer mapping to this new data types gets necessary. This concerns, for example, **sequence**, **sequence of**, **enumerated**, and **choice** which are used as **record**, **record of**, **enumeration**, and **union**. Hence, TTCN-2 mapping can mostly be used for TTCN-3. Their mapping is detailed below and summarised in Table 5.4 on page 93.

Struct

Type **struct** is used to collect data in one place. Because of the importance of ordering inside **struct**, it is always mapped to ASN.1 **sequence**. This ordering is until now not mentioned in the IDL specification but it is indirectly mentioned in the CORBA standard, for instance, in the *Internet Inter*-ORB *Protocol* (IIOP) specification and *type code* specification (OMG 2001b, chapter 15). Therefore, mapping onto the new data type **record** in TTCN-3 is used.

IDL

```
typedef struct NC {
    string id;
    string kind;
} NameComponent;
```

TTCN-3

```
type record NameComponent {
    string id,
    string kind
}
```

Discriminated Union

Unions can store different types in one place but only one at the same time. In IDL, *unions* are discriminated to determine the actual type. Therefore, a **record** type is used, which contains two members. The first one stores the discriminator information using an enumeration type. The second member is a TTCN-3 **union** type whose members are defined according to the specified IDL **union** members.

IDL

```
union MyUnion switch( long ) {
    case 0 : boolean b;
    case 1 : char c;
    case 2 : octet o;
    case 3 : short s;
};
```

TTCN-3

```
type union MyUnionType {
    boolean b,          iso8859string c,
    octetstring o,      short s }

type enumerated MyUnionEnumType {
    boolean_b,          iso8859string_c,
    octetstring_o ,     short_s }

type record MyUnion {
    MyUnionEnumType kind,
    MyUnionType value_ }
```

Enumerations

In both languages *enumerations* can be used on the same way.

IDL

```
enum NotFoundReason {
    missing_node,
    not_context,
    not_object
};
```

TTCN-3

```
type enumerated NotFoundReason {
    missing_node,
    not_context,
    not_object
}
```

The constructed type mappings are summarised in Table 5.4.

Table 5.4.: IDL *to* TTCN-3 *mapping for constructed types*

IDL	*TTCN-2/ASN.1*	*TTCN-3*
struct	SEQUENCE	record
enum	ENUMERATED	enumerated
union	SEQUENCE	record with union and enumerated

5.4.3. Template Types

IDL supports the *template* types **sequence**, **string**, **wide string**, and **fixed** type. Their mapping is detailed below and summarised in Table 5.5 on page 95.

Sequence

IDL **sequence** is used to provide support for one-dimensional arrays with a fixed maximum size and a defined length. In contradiction to fixed arrays (see subsection 5.4.4 on page 95) only the valid sequence entries will be transmitted and the empty entries will not. Therefore, use of *unbounded sequences* is also possible. This makes a mapping onto arrays in TTCN-3 impossible wherefore only **set of** and **record of** are available. To keep the ordering of sequences the type **record of** has to be chosen to provide an appropriate mapping to TTCN-3.

IDL

```
typedef sequence<NameComponent> Name;
```

TTCN-3

```
type record of NameComponent Name;
```

String and Wstring

Types **string** and **wstring** are *sequences* of **char** and **wchar**. In TTCN-2, several different predefined character string types were defined which are all replaced in TTCN-3 by the two types **charstring** and **universal charstring**. As character encoding **charstring** uses ISO/IEC 646 and **universal charstring** uses ISO/IEC 10646. Since the introduction of *useful types* the charstring mapping can be more exact by using *useful type* **iso8859string** (see Table 2.2 on page 26). Therefore, **string** and **wstring** are mapped to **iso8859string** and **universal charstring**.

IDL

```
const string  name =       "My String";
const wstring wideName = "My String";
```

TTCN-3

```
const iso8859string  name :=  "My String";
const universal charstring
                     wideName := "My String";
```

Fixed Type

The **fixed** type represents a fixed-point decimal number. There was no corresponding type in TTCN-3 available but based on the following discussion an **IDLfixed** useful type was introduced into TTCN-3 (ETSI 2002a, appendix E.2.3.0).

If we use a **float** with an extension attribute containing the digit and scale the user cannot use this information in his program. Since the similarities between TTCN-3 and C the solution of the C language mapping of IDL (OMG 1999, section 1.14) is used. It maps the type **fixed** to a type **struct** which stores the number in a **char** array and the digit and scale number in extra variables. In TTCN-3, this can be realised by a **record** containing a **charstring** to store the number, and two integers for the digit and scale. Hence, the user can access scale and digit, too. A **record** is preferred against a **set** because of the initialiser notation for records which makes the use of **fixed** types under TTCN-3 more convenient. See the definition of **IDLUnsignedShort** and **IDLShort** on page 88.

IDL

```
typedef fixed<12,7> Fix;
```

TTCN-3

```
type record IDLFixed {
    IDLUnsignedShort digits,
    IDLShort         scale,
    charstring       value_
}

var IDLfixed  fix  :=
     { 12, 7, "12345.1234567" };
```

The template type mappings are summarised in Table 5.5.

5.4.4. Complex Types

The last kind of type declarators are the complex types **array** and **native**. Their mapping is detailed below and summarised in Table 5.6 on page 95.

[1] Mapping of this type was not considered.

Table 5.5.: IDL *to* TTCN-3 *mapping for template types*

IDL	*TTCN-2/ASN.1*	*TTCN-3*
sequence	SEQUENCE OF	record of
string	GraphicString, IA5String	iso8859string
wstring	GraphicString, BMPString	universal charstring
fixed	—[1]	record

Arrays

IDL **array** can be mapped directly to the TTCN-3 **array** type because they provide the same functionality.

IDL

```
typedef long NumberList[100];
```

TTCN-3

```
var long NumberList[100];
```

Native Types

The **native** type is used to allow implementation dependent types. TTCN-3 provides the type **address** to address entitities inside a SUT. Hence, **address** can be used for mapping of **native** and concrete implementation is left to the user. However, this type should not be used in service and application interfaces. It is mainly designed for internal use in CORBA itself.

IDL

```
native MyNativeVariable;
```

TTCN-3

```
address MyNativeVariable;
```

The complex type mappings are summarised in Table 5.6.

Table 5.6.: IDL *to* TTCN-3 *mapping for complex types*

IDL	*TTCN-2/ASN.1*	*TTCN-3*
array	SEQUENCE SIZE(n) OF	array
native	—[1]	address

[1] Mapping of this type was not considered.

5.5. Communication Declaration

The main purpose of IDL is the description of object interfaces wherefore the provided *attributes* and *methods*, in IDL called *operations*, are described. Each operation may raise *exceptions*. Operations, attributes, and exceptions are the only way provided by IDL to communicate with an object. Thus, operations and attributes can only be defined in interfaces. Exceptions can only be used by operations but can also be defined outside of interfaces. Their mapping is detailed below and summarised in Table 5.7 on page 102.

5.5.1. Operations

Apart from attributes, operations are the main part of interface definitions in IDL (see subsection 5.3.2) and are used, for instance, in the CORBA scheme as *procedures* which can be called by clients. Procedure calls in general are supported by TTCN-3 by means of synchronous communication operations which are used in combination with ports. In TTCN-3 procedures are defined by **signatures** (see subsection 2.4.4). Operations under IDL consist of an *invocation* semantics by an operation attribute, *return* results, an *identifier*, a *parameter* list, an optional *exception* expression, and an optional *context* expression. The mapping of all this parts to TTCN-3 will be described now.

Operation Attribute

IDL supports an optional **oneway** attribute for operations which implies best-effort invocation semantics without a guarantee of delivery but with a most-once invocation semantics. Oneway operations have to provide no **out** parameters and the return type **void**. Furthermore, no **raise** expressions are allowed but standard exceptions can still be raised. If no attribute is given the invocation semantics is at-most-once in case of an exception and exactly-once if it returns successfully.

Messages or *procedures* could be used for oneway operations because both would be a valid mapping from IDL perspective. However, the use of procedure-based ports for oneway operations is recommended because the IDL specification does not guarantee that oneway calls are non-blocking or asynchronous. Furthermore, CORBA implements oneway operations by synchronous communication, too.

Hence, best mapping of both kind of operations is given by use of synchronous communication wherefore no distinction for oneway operations gets necessary. However, it is not the same for CORBA wherefore the IDL information in the *interface repository* could be used to detect oneway operations and to handle them appropriate. However, this is only for CORBA-based SUTs possible. Furthermore, the **variant** attribute could be used to mark **oneway** operations which should be preferred.

By the way, CORBA supports also the *Asynchronous Method Invocation* (AMI) to provide non-blocking requests. This is realised by a client-side mapping which requires no or only small server side modifications. Therefore, special methods will be produced in addition to provide the new functionality. This new methods still use the synchronous communication mechanism wherefore no modifications in IDL have been necessary. However, there was a new IDL introduced called *implied IDL* which is generated from the IDL specification with the additional operations for the client side and is used to generate the client implementation stubs.

Introduction of AMI leads to the fact that a client is also a server if the *callback* model is used. This model requires a reply handler which is invoked by the server. Hence, mapping of IDL to TTCN-3 gets necessary from client and server perspective. Therefore, operation exceptions, as defined in subsection 5.5.3, can also be raised by the tester.

To sum up, oneway operations are best mapped to procedures and marked as oneway operation by a **variant** attribute. Based on the discussion above the **noblock** attribute for procedures has been introduced into TTCN-3 (ETSI 2002a, section 13.1) to support non-blocking procedure calls. Use of non-blocking or blocking procedures for oneway operations is left to the user.

Parameter Declarations

The parameter attributes **in**, **inout**, and **out** describe the transmission direction of parameters and can be mapped directly to the communication parameter attributes in TTCN-3 because they have exactly the same semantics.

The **return**, **out**, and **inout** parameters of operations have to be catched by a **getreply** statement in TTCN-3 which should be given in the call context.

Raises Expressions

A **raise** expression specifies all exceptions which can be thrown by an operation. It can be mapped directly to TTCN-3 because it can be indicated by the procedure signature definition by specifying an exception. Nevertheless, each operation can trigger a standard exception.

Context Expressions

A **context** expression provides access to local properties of the called operation. These properties consist of a name and a **string** value. The **context** expression can be mapped by redefining the operation with the **context** parameters included in the operation parameters (OMG 2001b, section 4.6). This is done in a TTCN-2 mapping by introducing an additional array parameter (Mednonogov 2000; Mednonogov et al. 2000). The additional parameter should be of type **array** containing a type **record** for each context parameter. The **record** itself contains two variables of type **string** for the context name and value.

An example mapping is demonstrated below. The TTCN-3 mapping part is divided into an operation definition and usage part. The definition part illustrates the mapping and the usage part is only introduced to demonstrate the usefulness of the mapping.

IDL

```
// not found exception is defined in section ''exception declaration''

string remoteProc1( in long Par11, out long Par12, inout string name1 )
     raises ( NotFound ) context( ''MyContext1'' );

// oneway procedure: no return value and no inout or out allowed!!!
oneway void remoteProc2( in long Par21, in long Par22, in string name2 );
```

TTCN-3

Operation definition

```
type record IDLContextElement {
    iso8859string name,
    iso8859string value_
}

type record of IDLContextElement IDLContext;

signature RemoteProcSignature1(
            in long Par11, out long Par12,
```

```
            inout iso8859string name1, in IDLContext context )
        return iso8859string
        exception( NotFoundException );

signature RemoteProcSignature2(
            in long Par21, in long Par22, in iso8859string name2 )
        with { variant "IDL:oneway FORMAL/01—12—01 v.2.6" };

type port RemoteProcPort procedure {
        out RemoteProcSignature1;
        out RemoteProcSignature2
}

type component SUT {
    port RemoteProcPort PCO
}
```

Operation usage

```
// not found exception is defined in section ''exception declaration''

var IDLContextElement contextElement := {
        name := "MyContext1", value_ := "MyContextValue" };
var IDLContext idlContext := { contextElement };

template RemoteProcSignature1 RemoteProcTemplate1 := {
    Par11 := 0,
    Par12 := 1,
    name1 := "my name",
    context := idlContext
}

template RemoteProcSignature2 RemoteProcTemplate2 := {
    Par21 := 2,
    Par22 := 3,
    name2 := "my other name"
}

var SUT myCorbaSystem := SUT.create;
connect(self:myPCO, MyCorbaSystem:PCO);
myCorbaSystem.start;

myPCO.call( RemoteProcTemplate1 ) {
    [] myPCO.getreply(RemoteProcSignature1:{—,*,*} value *) —> value MyResult1
                    param(MyPar12, MyName1) sender MySender1 {}
    [] myPCO.catch( RemoteProcSignature1,
                MyNotFoundExceptionTemplate ) {
            verdict.set( fail );
            stop;
        }
}
```

```
myPCO.call( RemoteProcTemplate2 );

// raising an exception can be done on this way but it is not used
// because only the SUT should raise this exception!!!
var NotFoundException myNotFoundException := {
 why            := missing_node,
 rest_of_name := "noname"

}

myPCO.raise( RemoteProcSignature1, myNotFoundException );
```

5.5.2. Attributes

An interface attribute is like a *set*- and *get*-operation pair to access a value. If an attribute is marked as **readonly**, the *get*-operation is used only. Therefore, attribute mapping can be done by the operation mapping as described in subsection 5.5.1 on page 96.

IDL

```
attribute string object_type;
```

TTCN-3

```
signature RemoteAttribGetSignature() return iso8859string;
signature RemoteAttribSetSignature( in iso8859string object_type );

type port RemoteProcPort procedure {
      out RemoteAttribGetSignature;
      out RemoteAttribSetSignature
}

type component SUT {
      port RemoteProcPort PCO
}

var SUT myCorbaSystem := SUT.create;
connect(self:myPCO, myCorbaSystem:PCO);
myCorbaSystem.start();

// get
myPCO.call() {
    [] myPCO.getreply( value * ) -> value MyResult {}

       // catch all exceptions
    [] myPCO.catch {
```

```
            verdict.set( fail );
            stop;
        }
}

// set
template RemoteAttribSetSignature RemoteAttribSetSignatureTemplate := {
    object_type := "my name"
}

myPCO.call( RemoteAttribSetSignatureTemplate );
}
```

5.5.3. Exceptions

In IDL, exceptions are used in conjunction with operations to handle exceptional conditions during an operation call. Thus, a special *struct*-like **exception** type is provided which has to be associated with each operation that can trigger this exception. TTCN-3 also supports the use of exceptions with procedure calls by binding it to **signature** definitions. However, it does not provide a special **exception** type. Hence, exceptions are defined as **struct** by using **record**. TTCN-2 has no support of exceptions. Thus, it is realised by using an PCO with asynchronous communication to get exception information from the test system.

The part exception *definition* is shown in the following example and use of exception binding in signature definitions and exception *catching* is shown in context of operation declaration in subsection 5.5.1 on page 98.

IDL

```
exception NotFoundException { NotFoundReason why; Name rest_of_name; };
```

TTCN-3

```
// definition of an exception type
type record NotFoundException { NotFoundReason why, Name rest_of_name }

// definition of a template for the defined exception type
template NotFoundException NotFoundExceptionTemplate (
  NotFoundReason par1, Name name) := {
    why := missing_node,
    rest_of_name := name
}
```

The interface mapping inclusive operations, attributes, and exceptions is summarised in Table 5.7.

Table 5.7.: IDL *to* TTCN-3 *mapping for interface elements*

IDL		*TTCN-2/ASN.1*	*TTCN-3*
operation		ASP	signature
attribute		ASP pair	signature pair
	readonly	ASP	signature
raise expression		CHOICE	signature exception option
context expression		—	additional signature parameter
interface	name space		group
	parameter	IA5String	address
	communication	PCO	port

5.6. Names and Scoping

The name definition scheme of IDL does not collide with the name definition in TTCN-3. Scoping is more restrictive in IDL than in TTCN-3 wherefore the IDL scoping rules have to be mapped appropriately to allow seamless mapping. IDL uses nested scopes for modules, interfaces, structures, unions, operations, and exceptions. Identifiers are scoped in types, constants, enumeration values, exceptions, interfaces, attributes, and operations. The hierarchical scopes in TTCN-3 are **module**, control part of **module**, **function**, **testcase**, and statement blocks within control part of **module**, **function**, and **testcase** .

Furthermore, TTCN-3 does not support overloading of identifiers so that no identifier name can be used more than once in a scope hierarchy. However, IDL allows redefinition of self defined types if defined inside a **module**, **interface**, or **valuetype**. Hence, identifiers have to be mapped by using their *path name* including all **interface** and **valuetype** names as designated in IDL and TTCN-3. The use of module names is not necessary because they are reflected by the TTCN-3 module structure. An underscore is used as a separator and existing underscores are doubled. Use of path names was also suggested by the TTCN-2 mappings.

To indicate the special treatment of *all* TTCN-3 statements derived from IDL, TTCN-2 mappings were using special naming schemes. This simplifies coding and type differentiation. TTCN-3 provides a new mechanism to attach attributes to language elements. The use of attributes makes code more readable and requires no special naming scheme. Therefore, the **variant** attribute can be used to indicate derivation of types from IDL

and special treatment for encoding by the test system. This was explicitly mentioned in section 5.4 on page 86 for data types because they require always correct encoding mechanism. This is used in TTCN-3 for the *useful type* **IDLfixed:**

TTCN-3

```
type record IDLfixed {
  unsignedshort digits ,
  short scale ,
  charstring value_ } with { variant "IDL:fixed FORMAL/01—12—01 v.2.6" };
```

An own naming scheme to tag all types is required. It was suggested to use the type name and if necessary add special attributes like **valuetype** via the keyword **and**. However, among others the discussion here led to the introduction of the **variant** attribute and a naming scheme for the IDL mapping was introduced by defining the useful type **IDLfixed**. Hence, the naming scheme for *variants* uses the word IDL, the statement type, and a standard reference with a version number as shown before.

Names of new types which are specially defined for the IDL mapping and their use in conjunction with IDL shall always begin with the word **IDL** to provide better distinction.

5.7. Summary and Outlook

Summary

The chapter is primarily focused on the definition of a set of OMG IDL to TTCN-3 mapping rules, in order to support adequate testing based on given UML class diagram specifications. The presented mapping rules have been used to implement a converter to achieve semi-automated development of test data specifications. To support automated generation of the dynamic part in an ATS, additional formalised behavioural descriptions, e.g. sequence diagrams, have to be considered as shown in chapter 4. The mapping is based on some recent work in Ebner (2001a, b); Ebner et al. (2002); Yin (2001); Yin et al. (2001). It is also summarised in the ETSI Technical Specification 102 219 (ETSI 2003). A concept mapping list is given in Table A.1 on page 121. A comparison of IDL, ASN.1, TTCN-2, and TTCN-3 data types is given in Table A.2 on page 123.

It was not always possible to provide satisfiable mappings because mapping for **interface** and **valuetype** is not powerful enough in order to use all IDL features in a proper way. Nevertheless, this is no limitation in usage

but a matter of convenience. Furthermore, some mappings could not be very exact because of lacking support in TTCN-3, for instance, **double** and **long double**. The latter ones can only be mapped to type **float** which has no exact size definitions. Thus, only the **variant** attribute contains the mapping information. Operations, attributes, and exceptions can be mapped well to TTCN-3 because synchronous communication was introduced especially for this purpose.

To provide a better mapping of IDL and to enhance usability of TTCN-3 some elements have been especially introduced or extended as proposed by the author:

- The **variant** attribute, inclusive a set of predefined variants, was introduced. They are used to define *useful types* for integers, floats, strings, fixed, etc. as shown in ETSI (2002a, appendix E) and listed in Table 2.2 on page 26.
- The **anytype** was especially introduced for an exact mapping.
- Non-blocking procedure calls have been added.

Implementation specific parts like variable handling if used as **inout** parameters are not considered in TTCN-3 because it should be done by the compiler or interpreter and/or the underlying test system. Nevertheless, if IDL is not precise enough, e.g., the **oneway** attribute for operations, the CORBA specification was used as reference.

There was no discussion about technical implementation details like memory management, size limitations, and transmission by value or reference. However, it has been proven that TTCN-2 is generally applicable to the testing of CORBA-based systems (Li et al. 1999; Mednonogov 2000; Mednonogov et al. 2000; Schieferdecker et al. 1998). Furthermore, TTCN-3 gives support for synchronous communication wherefore implementing test suites gets easier, too. Hence, the implementation of the above mapping is feasible. If TTCN-3 is used to test CORBA systems, the language mapping of TTCN-3 should be equivalent to the IDL language mapping. In contrast to TTCN-2, mapping to TTCN-3 is more comfortable and powerful.

The IDL mapping is also useful in conjunction with UML and XML. In context of UML the mapping is useful because many UML tools provide exporting facilities to convert data from UML models to IDL. Furthermore, XMI (OMG 2003e) supports transfer of UML models and IDL by usage of XML. A mapping of IDL to WSDL respectively *Simple Object Access Protocol* (SOAP) and vice versa can be found in OMG (2003a, d). In

OMG (2003f) a description is given to create a data structure of XML documents to pass it in CORBA interface operations. The SODL is an XML IDL DTD which allows objects to be described in compatibility with IDL used in *Component Object Model* (COM) and CORBA. SODL is used in XML *Metadata Object Persistence* (XMOP) which is an object serialisation mechanism.

Outlook

Future work should address an adaptation to the newest IDL standard. The support of XML by TTCN-3 gets important wherefore this direction also has to be followed.

The presence of object-orientation was missed during developing the mapping. Thus, some improvements to TTCN-3 could be made if *inheritance* and *abstraction* as given by object-oriented concepts gets introduced. This would improve maintenance and handling of test suites and the mapping of IDL could be much clearer (see chapter 6).

6. Object-Oriented Enhancements for TTCN-3

Although TTCN-3 has reached a certain stage of maturity since its first release in 2001, its language elements are still lacking support of object-orientation. It is believed that object-orientation would ease its usage and make it more expressive and applicable. Especially in conjunction with object-oriented languages, platforms, or models like C++ and JAVA, the CORBA, and the UML.

Until now, TTCN-3 is not intended to support object-orientation except the support for synchronous communication for CORBA-based systems. However, object-oriented concepts have been missed during developing the IDL to TTCN-3 mapping as described in chapter 5. Especially, *inheritance* and *abstraction* concepts have been missed. They would improve maintenance and handling of test suites and simplify the mapping of IDL. There is ongoing work on the integration of TTCN-3 and UML by UTP (Schieferdecker et al. 2003) and a mapping of UTP to TTCN-3 by ITU-T standard Z.149. The deficiency of object-orientation in TTCN-3 is also mentioned in Schmitt (2003, page 41).

It is not intended to detail how to test object-oriented systems but the necessities to make TTCN-3 itself object-oriented. Testing object-oriented systems are discussed in section 2.2 and, for instance, in Binder (2000).

The remainder of this chapter is structured as follows. Firstly, TTCN-3 is inspected to take stock object-oriented concepts or systems like them and where it would be useful to have those concepts. Secondly, a concept of an object-oriented revision of TTCN-3 is suggested which is based on the preceding inspection. Finally, a summary and an outlook are given.

6.1. Object-Orientation in TTCN-3

Proposing an object-oriented enhancement of TTCN-3 requires an inspection to take stock object-oriented approaches. Furthermore, the deficiency of object-orientation and where it would be useful have to be identified. Not directly related object-oriented concepts will be mentioned anyhow if they can still be improved. Based on the inspection an enhancement is worked out which is given in section 6.2.

TTCN-3 was not especially designed to support object-oriented concepts or to test object-oriented systems wherefore no direct support is given. Nevertheless, the application of testing CORBA-based platforms is explicitly mentioned, but it is only supported in sense of communication operations such as procedure-based communication with the well known parameter attributes **in**, **inout**, and **out** from CORBA resp. IDL.

Furthermore, TTCN-3 uses the term *object* (see ETSI 2002a, section 5.4) and an object-oriented syntax like the well known dot notation to access operations, although there is no object concept given.

The used object-oriented concepts, the influence of object-oriented languages, and the deficiency of object-orientation will be shown in detail below. Furthermore, we mention among others the **exception** and **goto** concept.

Scope and Identifier

Scoping is used for **module**, *control part* of modules, **component** types, **function**, **altstep**, **testcase**, and *statement* blocks in compound statements. However, *groups* and *types* other than **component** have no own scoping wherefore only a little data encapsulation is provided. Hence, structuring test suites on a syntactical level is provided which can be used only by the test suite user. However, group attributes are forwarded to all statements in the group. What can be seen as a kind of *inheritance*. Group identifiers require not to be globally unique.

Nevertheless, the usage of local variables, constants, timers, and ports that are declared in a component type definition is interesting. To use these elements in a function, the **runs on** clause is required to state that this function can only operate in the given **component**. Hence, an explicit declaration has to be made to get access to the **component** scope.

Identifier shall be unique in a scope hierarchy wherefore no local variable with the same identifier as a global variable can be used. Hence, it is not allowed to overwrite global variables by local variables. Identifiers for fields of structured types require not to be globally unique.

Parameters

In TTCN-3, the type of parameter passing is defined by additional attributes. For this purpose, the parameter attributes **in**, **inout**, and **out** from CORBA resp. IDL are used. The attribute **in** means *passing by value* resp.

call by value and the attributes **inout** and **out** mean *passing by reference* resp. *call by reference*.

Attributes

The support of attributes to language elements can be seen as an extension or specialisation why it is comparable to the concept of inheritance from a basic language element which provides this attributes. However, the concept of object-orientation is applied to types and not intended for language elements wherefore only the attributes **encode** and **variant** could be replaced by using an object-oriented concept.

Modules and Groups

Modules are the top-level structuring element in TTCN-3 and they consist of a *definition* and *control* part.

The definition part is comparable to a class where all definitions are given. In the control part, test case execution order is given why it can be seen as the *main* method of the module. Modules can *import* definitions from other modules which allows to provide a kind of *inheritance*. Nevertheless, there are no global variables, timers, etc. available. Hence, modules are like packages in JAVA or name spaces in C++ but on the other hand they work like classes.

Groups are used to structure test data. To access elements in the group hierarchy the dot notation is used. There are no language constructs available to control the execution of test cases within groups.

Unfortunately, modules cannot be nested. Definitions can be combined into groups but a group does not define a new scope and has no semantic purpose except when definitions are imported by another module. Moreover, TTCN-3 assumes that the entire test execution is controlled exclusively by the control part of the current module and some auxiliary functions. Thus, test cases structured by groups cannot be explicitly combined with their execution order. Hence, its control part seems to be too inflexible for complex and large-scale test suites.

Test case execution inside a module can only be structured by cascading function calls. Thus, for each module and group, that has to be structured, one function has to be defined to control test case execution of all test cases inside (Schmitt & Ebner 2003).

Data Types

Many TTCN-3 data types are based on ASN.1 data types (ITU-T 1997c) as was done before in TTCN-2. The ASN.1 data type **objid** uses the term *object* but it can only be used to store ASN.1 object identifiers and has nothing to do with any TTCN-3 type. Hence, it is not the general *object* type which is necessary in each object-oriented language.

The fields of the structured types **record** and **set** are referenced by the dot notation as by objects. They can be seen as a kind of a public class type like the well known **struct** type. The **union** type uses also the dot notation to access its fields.

There is no class type resp. object type defined in TTCN-3 and consequently, there are only the structured types (**record**, **record of**, **set**, **set of**, **enumerated**, and **union**) available to structure data (see ETSI 2002a, section 6.3). Furthermore, the pre-defined functions for handling and converting basic types are stand-alone functions which are not bound to their corresponding types (see ETSI 2002a, appendix C). Hence, better structuring of data and predefined functions could be provided by usage of classes.

Signatures

TTCN-3 especially supports procedure-based communication to be applicable for object-oriented systems. For procedure-based communication, the definition of procedure signatures is required which comprises the blocking characteristic, parameters (see page 108), a return value, and exceptions.

Due to the requirement of TTCN-3 to support testing of CORBA-based systems the non-blocking characteristic was introduced. This was based on the discussion in subsection 5.5.1 on page 96. CORBA resp. IDL allow the definition of **oneway** procedures which do not block execution because they return immediately after invocation (see ETSI 2003, chapter 10).

Templates

Templates are used to organise and to re-use test data. They can be seen as instances of classes which provide concrete values for sending data to the SUT or which provide matching mechanisms for testing against received data. Templates provide the possibility to use a simple form of *inheritance* and they can be parametrised. Furthermore, in TTCN-3 the concept of inheritance is realised in templates only.

Test Configuration

> TTCN-3 is used to test implementations. The object being tested is known as the Implementation Under Test or IUT (ETSI 2002a, section 8.3).

Hence, it is common to see the SUT resp. *Implementation Under Test* (IUT) as an *object* which has to be tested. However, the model describing the configuration is motivated by the view that is used in the CTMF and consequently, in TTCN-2, too. The model uses components and ports to arrange tests and to describe the communication.

Components have local constants, variables, and timers and ports are described by signatures. *Polymorphism* can be realised by usage of optional fields in templates to send data and by usage of matching mechanism to receive data. Hence, they have object-oriented characteristics but there is no *inheritance* and no own scope available. There is a component reference available in TTCN-3 but it can only be used to bind communication statements to a concrete component and it is not allowed to use it as a parameter for communication operations. Ports cannot be used as a parameter, too.

TTCN-3 provides the open type **address** to access entities inside the SUT. The actual data representation of type **address** can be left open or can be set by an explicit type definition. However, it is not very convenient to use it for object references because it is not possible to directly call a method of the referenced object. Furthermore, if an object has to be send to or received by the IUT it has to be masked, for instance, by a **record** type containing the values of the *public* variables inside the object.

Hence, usage of object references would be very useful for TTCN-3 itself and to easier definition of ATSIs to test object-oriented systems. Concrete problem descriptions can be seen in the mapping of IDL to TTCN-3 as done in chapter 5.

Function, Altstep, and Testcase

Functions, *altsteps*, and *testcases* can have the optional attribute **runs on** to state the dependency of a component. Hence, the **runs on** clause is a means to separate the definition of a component and its depending functions, altsteps, and testcases. Consequently, they should be defined or at least declared together as it is done with classes to state this dependency more clearly.

Table 6.1.: *Corresponding data types for operations in* TTCN-3

Operations	*Corresponding Type*
Configuration	component
Communication	port
Timer	timer
Verdict	component

There is no explicit *polymorphism* for functions, altsteps, and testcases available as is the case for signatures. However, it can still be realised by the usage of type **record** and templates but an explicit solution would be more expressive and much clearer.

Altsteps provide local variables, parameters, and a kind of *inheritance*. Furthermore, they can be activated at the beginning of a test case and they will automatically be used in each following **altstep** statement. Therefore, altsteps are very good candidates to provide an own class for it.

Program Statements

The usage of program statements is not directly related to object-orientation but it is also useful for the application of TTCN-3. Hence, it is discussed here, too.

Although exceptions can be used in signatures for procedure-based communication the usage of exceptions in the test cases themselves is not possible. It might be that the program complexity of test cases is not that high but nevertheless it could be useful.

The usage of **label** and **goto** makes reading of test cases more difficult and produces well known problems. Thus, it should be replaced.

Operations

All configuration, communication, timer, and test verdict operations are bound to an instance of the corresponding types (see Table 6.1). For operation calls the dot notation is used. Therefore, all the types can be seen as a kind of class with given methods.

Configuration The configuration operations **mtc** and **system** return a component reference wherefore they can be seen as class variables. The standalone operation **self** is bind implicitly to the component in which it is

called. Therefore, **self** works like the keyword **this** in JAVA and C++ which returns the self reference, too.

Communication The operation **call** can deliver replies, timeouts, and exceptions. In case of non-blocking procedure-based communication they have to be handled in following alternative statements. For blocking procedure-based communication, an alternative statement is directly attached to the **call** statement. In conjunction with default handling via altsteps, an expressive way to handle timeouts and exceptions is provided for a test case. It is not only locally but also globally usable. However, exceptions are only available for **call** operations.

The feature to retrieve variable values from **receive**, **getcall**, and **getreply** operations is also interesting. In **receive** operations the received message value and sender address can be retrieved by assigning it to variables. The same can be done in **getcall** and **getreply** operations for the sender address and parameter values of a received call or reply. This is done in an optional assignment part of the operators and looks very similar to the component initialisation list of constructors in C++. The intention of TTCN-3 is to keep the necessary data only if required for further test case execution because many times the matching result itself is sufficient enough to make a decision.

Verdicts Each test component and test case provides a verdict *object* and the test case verdict depends on the component verdicts in the test case. TTCN-3 uses the explicit term *object* for the component verdict but it is not further defined and hence, left to the implementation. Furthermore, there is a type **verdicttype** where verdicts can be stored but the overwriting rules for verdicts are not valid for it. The overwriting rules are only used by the **setverdict** operation to set *local* verdicts. Hence, a variable of type **verdicttype** can contain any verdict at any time independent of former values and is mainly used to allow computation depending on verdicts.

6.2. Object-Oriented Revision of TTCN-3

As we could see in the inspection before, TTCN-3 provides expressive *parameter passing* (see page 108) and *procedure-based* communication mechanisms (see page 110). However, there is only simple support for *inheritance* (see pages 109 and 110) and *polymorphism* (see page 110) available. The structure of depending elements, like components and testcases,

could be improved by using classes, for instance. Furthermore, the type system provides no support for object-orientation, there are no hierarchical modules, and there are only a few matching mechanism for structured types available.

The object-oriented concepts mainly discussed below are *encapsulation*, *inheritance*, and *polymorphism*. The main revision purposes are to improve handling of TTCN-3 language elements itself and mapping of SUT structure and their communication description. Based on the preceding inspection a suggestion of a concept for an object-oriented revision of TTCN-3 gets presented now.

6.2.1. Data Types

For data types, no support for object-orientation is given and to provide seamless integration into TTCN-3 existing types will not be touched here. We introduce *classes*, which can be seen in the next subsection, and an **object** type to store instance references of classes. Hereby, the handling of class instances has to be considered, especially in context of parameter passing.

Class instances could be handled by *references*, *pointers*, or *objects* itself as it is possible in C++. However, to provide a seamless integration, using pointers is declined and references are preferred as intended by TTCN-3. The difference between a reference and an object itself is, for instance, noticeable by assignments or if used as a parameter (see page 108). As mentioned before, objects have to be interpreted as references if used as **inout** or **out** parameter and as object itself if used as **in** parameter. To test the validity of an object reference variable the literal **null** gets introduced which stands for *no reference assigned*. In addition, the keyword **this** gets introduced to get the self reference of an object.

Furthermore, supporting corresponding classes is suggested to collect pre-defined functions and permit the usage of basic types as objects. Additionally, sub-typing of basic types can also be realised by usage of inheritance. This concerns the types **integer**, **float**, **char**, **universal char**, **verdicttype**, and **union**. For all string types a common class **string** shall be defined. An own class for handling the useful type **IDLfixed** is suggested, too.

6.2.2. Classes

As stated before, classes get introduced into TTCN-3 to provide *encapsulation*, *inheritance*, and *polymorphism* which are basic concepts from

Table 6.2.: *Class element accessibility by access attributes*

Attribute	*Accessibility*
private	class
protected	class and subclasses
public	all (class, subclasses, and environment)

object-oriented languages. The new keyword **class** is used in defining a class. The three concepts will be discussed in detail below.

Encapsulation

With classes, it is possible to associate data and algorithms. Thus, an own scope and local constants, variables, and timers are provided as done by the type **component** (see page 111), too. Classes combine functionality from *groups* and *types* (see page 108) and could be seen as an extension of the type **component**. Furthermore, definitions of types, **signature**, **template**, **function**, **altstep**, and **testcase** in classes are supported. In case of **altstep** and **testcase** using the **runs on** clause gets redundant if a class is used as component. A class can be used as **component** if ports are given as class attributes. Hence, the dependency between a component and their corresponding **function**, **altstep**, and **testcase** definitions gets possible as demanded earlier (see page 111).

All definitions and declarations inside a class can get an *access* attribute **private**, **protected**, or **public** (default) to define the visibility to the environment and subclasses as shown in Table 6.2. Hence, type **component** is a special case of **class** where only ports and local constants, variables, and timers are allowed and all are declared **public**. This is comparable to types **struct** and **class** in C++. Access to class elements like attributes and functions shall be given by the dot notation (as used in C++) if the access attributes permit it.

To a class definition itself an *access* attribute **private** (default) or **public** can be given to state the visibility to the modul environment. Private classes are only visible inside the modul whereas public classes can be used from other modules. Classes imported by the **import** statement are considered as defined in the module itself.

In Schmitt (2003) and Schmitt & Ebner (2003) the replacement of the module control part by **control** functions was suggested. A **control** function

inside a class could be used to control execution of all defined test cases of the class.

Inheritance

Reuse can be achieved by inheritance where a class can inherit from other classes. In TTCN-3 the concept of inheritance is used by groups, modules, and templates. Inheritance for classes can supplement or even replace these existing concepts. Class inheritance is done by the new keyword **extends** and a comma separated list of classes from which the class inherits elements.

Attributes can be used for classes whereby the same inheritance mechanism as for groups is used. The usage of classes for **import** statements is the same as for other types. Nevertheless, using **class** inheritance and nested modules can make **import** statements obsolet. Inheritance for components respectively classes enables the support of generic component types which would improve structuring test suites. Additionally, *abstract* classes are introduced to support generic components or classes more clearly. Hence, the keyword **abstract** gets introduced to mark base classes which can only be used by inheritance and not by instantiation.

6.3. Summary and Outlook

Summary

The chapter discusses improving of object-orientation in TTCN-3 to widen its expressiveness and applicableness. TTCN-3 provides an object-oriented approach by using operations, signatures, and templates. It uses some object-oriented syntax like the dot notation, too. However, the concepts have to be improved and if we take a look on data types, object-orientation is still missing.

It was shown how basic object-orientation can be seamless and explicitly introduced to improve clarity of existing concepts and to widen the capabilities of TTCN-3. Therefore, the new type **object**, the keyword **this**, the literal **null**, and classes, including class attributes, have been introduced.

Outlook

In conjunction with hierarchical modules and a control function instead of a control part, as mentioned in Schmitt & Ebner (2003), the object-oriented revision of TTCN-3 could be given as proposal. Furthermore, the inspection and revision suggestion can be used to get a new view on TTCN-3, to use it as a basic work for the next generation of TTCN, and for the usage of TTCN-3 together with UML.

Nevertheless, beside an explicit syntax a deeper discussion about usage of the new object-oriented context has to be worked out. Especially in context of IDL and UML. Another further step would be a new object-oriented test description language which can be easily used in corporation with UML.

7. Conclusion

This thesis treated Computer Aided Test Generation based on manual graphical test purposes due to overcome limits of automatic test generation. Test case generation based on state space exploration is helpful but produces frequently inefficient test cases, lacks to cover specific parts, or may not be possible because of an incomplete or missing specification. Hence, manually defined graphical test purposes by MSC are desirable. Using UML for test purpose specification provides the advantage to use a well known language and to integrate given system specifications with test specification. In addition, given information like used types or interfaces by IDL are used for the static part of a test suite. Thus, a more complete test suite can be defined using UML and integration with automatic test generation is given. The industrial interest on the MSC to TTCN-3 prototype and the ETSI Technical Specification of the IDL to TTCN-3 mapping have shown the need of this thesis. The practicability was shown by prototypes which implement the mappings of MSC and IDL to TTCN-3.

An object-oriented extension of TTCN-3 was only indicated and should be considered for further improvements of TTCN-3 to widen its usability.

A. IDL Mapping Summary

A.1. Conceptual IDL to TTCN-3 Mapping

Table A.1 lists the mapping of keywords and concepts of IDL to TTCN-3 keywords or concepts. Literal mapping can be seen in Table 5.1 on page 81 and operator mapping in Table 5.2 on page 86.

Table A.1.: *Conceptual list of* IDL *mapping*

IDL	*TTCN-3*
FALSE	false
Object	address
TRUE	true
abstract	has to be rolled out
any	anytype
array	array
attribute	get (and set) operation
boolean	boolean
char	iso8859char
	(self defined type)
const	const
context	additional procedure
	parameter of type record
enum	enumerated
exception	record
fixed	IDLfixed
float	IEEE754float
double	IEEE754double
long double	IEEE754extdouble
in	in
inout	inout
interface	group, port
local	—
long	long
long long	longlong

IDL	*TTCN-3*
module	module
native	address
octet	octetstring
oneway	operation with
	variant attribute
operation	signature for procedure
out	out
raises	exception
readonly	only a get operation
	for the attribute
sequence	record of
short	short
string	iso8859string
struct	record
typedef	type
union	record, enumerated, union
unsigned long	unsignedlong
unsigned long long	unsignedlonglong
unsigned short	unsignedshort
valuetype	record
wchar	universal char
wstring	universal charstring

A.2. Comparison of IDL, ASN.1, TTCN-2, and TTCN-3 Data Types

Table A.2 lists a comparison of IDL, ASN.1, TTCN-2, and TTCN-3 data types and is based on the documents (Li 1998; Mednonogov 2000; Mednonogov et al. 2000; Open Group 2000) and is also summarised in ETSI (2003).

[1] Mapping of this type was not considered.
[2] superseded in ASN.1 from 1997 (ITU-T 1997b)

Table A.2.: *Comparison of* IDL, ASN.1, TTCN-2, *and* TTCN-3 *data types*

IDL	*ASN.1*	*TTCN-2*	*TTCN-3*
Object	ObjectInstance (X.500 Distinguished name)	IA5String	address
any	ANY DEFINED BY[2] or SEQUENCEtypecode, anyValue	CHOICE	anytype
array	SEQUENCE OF (with sizeConstraint subtype)	SEQUENCE SIZE(n) OF	array
boolean	BOOLEAN	BOOLEAN	boolean
char	GraphicString	GraphicString or IA5String(SIZE(1))	iso8859char (self defined type)
enum	ENUMERATED	ENUMERATED	enumerated
exception	SPECIFIC ERRORS	SEQUENCE	record
fixed	—[1]	—[1]	IDLfixed
float	REAL	—[1]	IEEE754float
double	REAL	—[1]	IEEE754double
long double	REAL	—[1]	IEEE754extdouble
long	INTEGER	INTEGER	long
long long	INTEGER	INTEGER	longlong
native	—[1]	—[1]	address
octet	OCTET STRING	OCTET STRING (SIZE(1))	octetstring
sequence	SEQUENCE OF (with optional sizeConstraint subtype for IDL bounds)	SEQUENCE OF	record of
short	INTEGER	INTEGER	short
string	GraphicString	GraphicString	iso8859string
struct	SEQUENCE	SEQUENCE	record
union, switch, case	CHOICE (with ASN.1 TAGS)	SEQUENCE	record, union, enumerated
unsigned long	INTEGER	INTEGER	unsignedlong
unsigned long long	INTEGER	INTEGER	unsignedlonglong
unsigned short	INTEGER	INTEGER	unsignedshort
valuetype	—[1]	—[1]	record
wchar	—[1]	GraphicString or BMPString(SIZE(1))	universal char
wstring	—[1]	GraphicString	universal charstring

A.3. Examples

A completion of all examples used in chapter 5 is given here. Some parts are used from the CORBA *Naming Service* with slight modifications to cover more IDL elements. The first part of the module definition part was generated by the implemented IDL to TTCN-3 converter. The second part demonstrates how the first part definitions can be used for test case definitions.

A.3.1. Given IDL Specification

```
module NamingServiceExample
{
  // ***********
  // Basic Types
  // ***********
  const long    number    = 017;  // 017 == 0xF == 15
  const long    size      = ( ( number << 3 ) % 0x1F ) & 0123;
  const float   decimal   = 15.7;

  const char    letter    =  'A';
  const wchar   wideLetter =  'A';

  const boolean isValid   = TRUE;
  const octet   anOctet   = 0x55; // limited to 8 bit

  const string  myName    =  "my name";
  const wstring wideMyName = "my name";


  typedef string  MyString;

  // *****************
  // Constructed Types
  // *****************
  typedef struct NC {
      MyString id;
      MyString kind;
  } NameComponent;

  union MyUnion switch( long ) {
      case 0 : boolean b;
      case 1 : char c;
      case 2 : octet o;
      case 3 : short s;
  };

  enum NotFoundReason { missing_node,
                        not_context,
                        not_object };

  // **************
  // Template Types
  // **************
  typedef sequence <NameComponent> Name;

  typedef sequence <NameComponent> Key;

  typedef fixed<12,7> Fix;

```

```
50    // ******************
51    // Complex Declarator
52    // ******************
53    typedef long NumberList[100];
54
55    native MyNativeVariable;
56
57    // ********************
58    // Valuetype Definition
59    // ********************
60
61    valuetype StringValue string;
62
63    valuetype EmployeeRecord {
64        // note this is not a CORBA::Object
65        // state definition
66        private string name;
67        private string email;
68        private string SSN;
69
70        // initializer
71        factory init (in string name, in string SSN);
72    };
73
74    // ********************
75    // Interface Definition
76    // ********************
77    interface NamingContext {
78        attribute string object_type;
79        readonly attribute Key external_form_id;
80
81        exception NotFound {
82            NotFoundReason why;
83            Name rest_of_name;
84        };
85
86        MyString bind( in Name n,
87                       inout Object obj,
88                       out Object myObj )
89            raises( NotFound )
90            context ( "Hostname" );
91
92        oneway void rebind( in Name n,
93                            in Object obj );
94
95    }; // end of interface NamingContext
96
97 }; // end of module
```

A.3.2. Mapped TTCN-3 Specification

Generated Part

```
// ********************************************************************
// Generated by the IDL to TTCN—3 translator
//
// Copyright (c) 2003
// Institute for Informatics, University of Goettingen
// 37083 Goettingen, Germany
//
// All Rights Reserved
//
// ********************************************************************

module NamingServiceExample() {
  const long number := '17'O;

  const long size := ( ( number << 3 ) mod '1F'H ) and4b '0123'O;

  const float decimal := 15.7;

  const iso8859char letter := "A";

  const universal char wideLetter := "A";

  const boolean isValid := true;

  const octetstring anOctet := '55'H;

  const iso8859string myName := "my name";

  const universal charstring wideMyName := "my name";

  type iso8859string MyString;

  // struct
  type record NC {
      MyString id,
      MyString kind
  };
  type NC NameComponent;
  // end struct

  // union
  type union MyUnionType {
      boolean b,
      iso8859char c,
      octetstring o,
      short s
  }
```

```
48  type enumerated MyUnionEnumType {
49      boolean_b,
50      iso8859char_c,
51      octetstring_o ,
52      short_s
53  }
54  type record MyUnion {
55      MyUnionEnumType kind,
56      MyUnionType value
57  }
58  // end union
59
60  // enum
61  type enumerated NotFoundReason {
62      missing_node,
63      not_context,
64      not_object
65  };
66  // end enum
67
68  type record of NameComponent Name;
69
70  type record of NameComponent Key;
71
72  type IDLfixed Fix;
73  template IDLfixed FixTemplate := {  12,  7, ? };
74
75  type long NumberList[ 100 ];
76
77  address MyNativeVariable;
78
79  group NamingContextInterface {
80      signature object_type ( )
81          return iso8859string;
82      signature object_type ( in iso8859string _object_type );
83
84
85      signature external_form_id ( )
86          return Key;
87
88
89      // exception
90      type record NotFound {
91          NotFoundReason why,
92          Name rest_of_name
93      };
94      template NotFound NotFoundTemplate (
95        NotFoundReason _why, Name _rest_of_name ) := {
96          why := _why,
97          rest_of_name := _rest_of_name
98      };
```

```
99        // end exception
100
101       signature bind (
102               inout Name n,
103               inout address obj,
104               out address myObj
105           )
106           return MyString
107           exception( NotFound );
108
109       signature rebind (
110               inout Name n,
111               inout address obj
112           ) noblock;
113
114       type port NamingContext
115           procedure {
116           object_type,
117           external_form_id,
118           bind,
119           rebind
120       }
121       type address NamingContextObject;
122   } // group NamingContextInterface
123
124 } // module NamingServiceExample
125
126 // ********************************************************************
127 // End of generated TTCN-3 code
128 // ********************************************************************
```

Test Case Example

```
module NamingServiceExample() {

  // somewhere has MyMTC to be defined

  type record IDLContextElement {
      iso8859string  name,
      iso8859string  value_
  }

  type record of IDLContextElement IDLContext;

  // ********************
  // Testcase  Definition
  // ********************
  testcase  MyNamingServiceTestCase() runs on MyMTC system CorbaSystemInterface {

      // examples to show how above defintions can be used inside a
      // testcase   definition

      var  CorbaSystemInterface myCorbaSystem := CorbaSystemInterface.create;
      connect( self :NamingContextPCO, myCorbaSystem:PCO );
      myCorbaSystem.start;

      //
      // Fixed Type
      //
      var IDLfixed  fix  := {  12,  7,  "12345.1234567" };

      //
      // Array
      //
      var integer  numberList[100];

      //
      // Native
      //
      var address MyNativeVariable;


      //
      // Procedure Calls
      //
      var iso8859string  myResult1;
      var Key            myResult2;
      var MyString   myResult3;
      var iso8859string  object,  myObject, resultObject,  resultMyObject;

      var IDLContextElement contextElement := {
          name := "Hostname",
```

```
50              value_ := "disen"
51          }
52
53          var IDLContext contextParameter := { contextElement };
54
55          //
56          // procedure get object_type
57          //
58          NamingContextPCO.call( ObjectTypeGetSignature )
59          {
60              [] NamingContextPCO.getreply( ObjectTypeGetSignature value * )
61                  -> value myResult1 {}
62          }
63
64          //
65          // procedure  set  object_type
66          //
67          NamingContextPCO.call( ObjectTypeSetSignatureTemplate );
68
69
70          //
71          // procedure get external_from_id
72          //
73          NamingContextPCO.call( ExternalFormIdGetSignature )
74          {
75              [] NamingContextPCO.getreply( ExternalFormIdGetSignature value * )
76                  -> value MyResult2 {}
77          }
78
79
80          //
81          // procedure bind (with template)
82          //
83          NamingContextPCO.call( BindTemplate( object, contextParameter ) )
84          {
85              [] NamingContextPCO.getreply( BindTemplate( * ) value * )
86                  -> value myResult3
87                  param( resultObject,  resultMYObject ) sender mySender {}
88
89              [] NamingContextPCO.catch( BindSignature,
90                                  NotFoundExceptionTemplate )
91              {
92                  verdict .set( fail );
93                  stop;
94              }
95
96          }
97
98
99          //
100         // procedure bind (without template)
```

```
101         //
102         NamingContextPCO.call(
103                         BindSignature:{  myName, object, myObject, contextParameter } )
104         {
105               [] NamingContextPCO.getreply( BindSignature:{ —, *, myObject }
106               value * )  —> value myResult3
107                     param( resultObject,  resultMYObject ) sender mySender {}
108         }
109
110         //
111         // procedure rebind
112         //
113         NamingContextPCO.call( RebindSignature:{ myName, object} );//or use a template
114
115
116         //
117         // raising an exception
118         //
119
120         // this would be used to raise an exception inside of procedure bind
121         // if defined by TTCN—3 (if used on server side).
122         var NotFoundException myNotFoundException := {
123               why             := missing_node,
124               rest_of_name := "noname"
125         }
126
127         NamingContextPCO.raise( BindSignature, myNotFoundException );
128
129     } // end of testcase MyNamingServiceTestCase
130
131 } // end of module
```

B. The TTCN-3 Inres Protocol Module

Below a directly defined TTCN-3 module for the *Inres protocol* is given (Schmitt 2003).[1]

```
1  /*
2   * TTCN—3 module for the 'Inres' protocol
3   *
4   * Copyright (C) 2002 Michael Schmitt <Michael.Schmitt@teststep.org>
5   *
6   * Date: 2002/09/10 18:47:37
7   */
8
9  module TestsForInres( integer maxRepetitions, boolean testInopportuneEvents ) {
10   import from ServiceUser language "ASN.1:1997" {
11     type UserPDU;
12     const someUserPDU;
13   }
14
15   group BasicDefinitions {
16     type UserPDU InresSDU; // the PDU on layer n+1 becomes an SDU on layer n
17
18     type enumerated InresPDUType { CR(1), CC(2), DR(3), DT(4), AK(5) };
19
20     type enumerated SequenceNumber { zero(0), one(1) };
21
22     type record InresPDU {
23       InresPDUType iPDUType,
24       SequenceNumber seqNo optional,
25       InresSDU iSDU optional
26     }
27
28     type InresPDU MediumSDU; // the PDU on layer n becomes an SDU on layer n—1
29   } with { encode "PER—BASIC—UNALIGNED:1997" } // apply Packed Encoding Rules
30
31   const float maxTestCaseTime := 50; // max. execution time for a single test case
32   const float maxTransferTime := 30; // max. execution time for a single data transfer
33
34   group CommunicationWithInitiator {
35     type record ICONreq {};
36     type record ICONconf {};
37     type record IDATreq { InresSDU iSDU };
38     type record IDISreq {};
```

[1] Copy permission was thankfully granted by Dr. Michael Schmitt.

```
39      type record IDISind  {};
40
41      type port  InitiatorSAP  message {
42        out ICONreq, IDATreq, IDISreq;    // sent to SUT
43        in  ICONconf, IDISind;        // received from SUT
44      }
45    }
46
47    group CommunicationWithMedium {
48      type record MDATreq { MediumSDU mSDU };
49      type record MDATind { MediumSDU mSDU };
50
51      type port MediumSAP message {
52        in MDATind;   // received from SUT
53        out MDATreq;  // sent to SUT
54      }
55    }
56
57    group CommunicationBetweenTestComponents {
58      signature acknowledgementSent();
59
60      type port PortAtMTC procedure {
61        in acknowledgementSent;
62      }
63
64      type port PortAtPTC procedure {
65        out acknowledgementSent;
66      }
67    }
68
69    group ComponentDefinitions {
70      type component MainTC {
71        port  InitiatorSAP  ISAP1;
72        port PortAtMTC CoordinationPTC;
73        timer supervisionTimer;
74      }
75
76      type component ParallelTC {
77        port MediumSAP MSAP2;
78        port PortAtPTC CoordinationMTC;
79      }
80
81      type component TestSystem {
82        port  InitiatorSAP  ISAP1;
83        port MediumSAP MSAP2;
84      }
85    }
86
87    group TemplateDefinitions {
88      template IDATreq InresDataRequest( InresSDU data ) := {
89        iSDU := data
```

```
90      }
91
92      template MDATind ConnectionRequest := {
93        mSDU := { iPDUType := CR, seqNo := omit, iSDU := omit }
94      }
95
96      template MediumSDU ConnectionConfirmation := { // this template is  used with
97        iPDUType := CC, seqNo := omit, iSDU := omit    // template 'MediumDataRequest'
98      }
99
100     template MDATreq MediumDataRequest( template MediumSDU data ) := {
101       mSDU := data
102     }
103
104     template MDATind DataTransfer( InresSDU data ) := {
105       mSDU := { iPDUType := DT, seqNo := ?, iSDU := data }
106     }
107
108     template MDATreq DataAcknowledgement( SequenceNumber number ) := {
109       mSDU := { iPDUType := AK, seqNo := number, iSDU := omit }
110     }
111   }
112
113   altstep  MTCFailure() runs on MainTC {
114     [] ISAP1.receive  {
115          setverdict ( fail );
116          stop;
117        }
118     [] any timer.timeout {
119          setverdict ( fail );
120          stop;
121        }
122   }
123
124   altstep  PTCFailure() runs on ParallelTC  {
125     [] MSAP2.receive {
126          setverdict ( fail );
127          stop;
128        }
129   }
130
131   altstep  ReceptionIDISind( verdicttype  result  ) runs on MainTC {
132     [] ISAP1.receive( IDISind : {} ) {
133          setverdict ( result );
134          stop;
135        }
136   }
137
138   function  MediumAccess() runs on ParallelTC {
139     var integer     receipt ;
140     var default     def :=  activate ( PTCFailure() );
```

```
141     var MDATind indication;
142
143     MSAP2.receive( ConnectionRequest );
144     receipt := 1;    // first (received) connection request of the initiator
145
146     MSAP2.send( MediumDataRequest( ConnectionConfirmation ) );
147
148     alt {
149       [ receipt <= maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
150             // connection confirmation got lost probably due
151             // to a malfunction of the medium; resend it
152             receipt := receipt + 1;
153             MSAP2.send( MediumDataRequest( { CC, omit, omit } ) );
154             repeat;
155           }
156       [ receipt > maxRepetitions ] MSAP2.receive( ConnectionRequest ) {
157             // even over an unreliable medium, the initiator
158             // shall not resend its requests that often
159             setverdict( fail );
160             stop;
161           }
162       [] MSAP2.receive( DataTransfer( someUserPDU ) ) -> value indication {
163             /* empty */
164           }
165     }
166
167     MSAP2.send( DataAcknowledgement( indication.mSDU.seqNo ) );
168
169     // inform the main test component that
170     // the data have been received and acknowledged
171     CoordinationMTC.call( acknowledgementSent : {} );
172     CoordinationMTC.getreply( acknowledgementSent : {} );
173
174     alt {
175       [] MSAP2.receive( MDATind : { mSDU := { DR, omit, omit } } ) {
176             setverdict( pass );              // disconnection request
177           }
178       [] MSAP2.receive( DataTransfer( someUserPDU ) ) {// data acknowl. got lost
179             setverdict( inconc );
180           }
181     }
182   }
183
184   testcase SingleDataTransfer() runs on MainTC system TestSystem {
185     var ParallelTC ptc;
186     var default def1, def2;
187
188     ptc := ParallelTC.create;
189
190     map( self:ISAP1, system:ISAP1 );
191     map( ptc:MSAP2, system:MSAP2 );
```

```
192
193     connect( self :CoordinationPTC, ptc:CoordinationMTC );
194
195     ptc. start ( MediumAccess() );
196
197     def1 := activate ( MTCFailure() );
198     def2 := activate ( ReceptionIDISind( inconc ) );
199
200     ISAP1.send( ICONreq : {} );          // connection request
201     ISAP1.receive( ICONconf : {} );      // connection confirmation
202
203     supervisionTimer. start ( maxTransferTime );   // restrict time for data transfer
204
205     ISAP1.send( InresDataRequest( someUserPDU ) );   // data transfer
206
207     // delay disconnection request until 'ptc' has received and acknowledged the data
208     CoordinationPTC.getcall( acknowledgementSent : {} );
209     CoordinationPTC.reply( acknowledgementSent : {} );
210
211     supervisionTimer.stop;    // cancel timer to avoid a timeout in the following
212
213     deactivate( def2 ); // a discon. indication is no undesirable event any longer
214
215     ISAP1.send( IDISreq : {} );          // disconnection request
216     ISAP1.receive( IDISind : {} );       // disconnection indication
217
218     all component.done;
219
220     setverdict ( pass );
221   }
222
223   testcase DataLoss() runs on MainTC system TestSystem {
224     // ...
225   }
226
227   control {
228     var verdicttype     overallVerdict := pass;
229
230     overallVerdict := execute( SingleDataTransfer (), maxTestCaseTime );
231
232     if ( overallVerdict == pass and testInopportuneEvents == true ) {
233       overallVerdict := execute( DataLoss() );
234     }
235   }
236 } with { encode "BER:1997" } // apply Basic Encoding Rules by default
```

Acronyms

A

AMI ..... Asynchronous Method Invocation
ASN.1 ..... Abstract Syntax Notation One
ASP ..... Abstract Service Primitive
ATS ..... Abstract Test Suite
ATSI ..... Abstract Test System Interface

C

CATG ..... Computer Aided Test Generation
CCITT ..... Consultative Committee for International Telegraph and Telephone (now ITU-T)
CORBA ..... Common Object Request Broker Architecture
CTMF ..... Conformance Testing Methodology and Framework

E

ETSI ..... European Telecommunications Standards Institute

F

FIFO ..... First In First Out

G

GFT ..... TTCN-3 Graphical Presentation Format

H

HMSC ..... High-Level Message Sequence Chart

I

IDL ..... Interface Definition Language
IEC ..... International Electrotechnical Commission
IIOP ..... Internet Inter-ORB Protocol
IOR ..... Interoperable Object Reference
ISO ..... International Organisation for Standardisation
ITU ..... International Telecommunication Union
ITU-T ..... ITU – Telecommunications Standardisation Sector (formerly CCITT)
IUT ..... Implementation Under Test

M

MSC ..... Message Sequence Chart
MTC ..... Main Test Component

O

OMG ..... Object Management Group
ORB ..... Object Request Broker
OSI ..... Open Systems Interconnection

P

PCO ..... Point of Control and Observation
PDU ..... Packet Data Unit
POA ..... Portable Object Adaptor
PTC ..... Parallel Test Component

R

RTSI ..... Real Test System Interface

S

SAP ..... Service Access Point
SDL ..... Specification and Description Language
SOAP ..... Simple Object Access Protocol
SODL ..... Simple Object Definition Language
SUT ..... System Under Test

T

TTCN ..... Tree and Tabular Combined Notation
TTCN-2 ..... Tree and Tabular Combined Notation (version 2)
TTCN-3 ..... Testing and Test Control Notation (version 3)

U

UML ..... Unified Modeling Language
UTP ..... UML Testing Profile

W

WSDL ..... Web Services Description Language

X

XMI ..... XML Metadata Interchange
XML ..... eXtensible Markup Language

Bibliography

Amyot, D. & A. Eberlein, 2003. An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunications Systems*, volume 24(1):pages 61–94. ISSN 1018-4864. Kluwer Academic Publishers.

Balzert, H., 1998. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, volume 2. Spektrum Akademie Verlag, 1 edition. ISBN 3-8274-0065-1.

Binder, R., 2000. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley. ISBN 0-201-80938-9.

Dai, Z., J. Grabowski, & H. Neukirchen, 2002. *TIMED*TTCN-3 – A Real-Time Extension for TTCN-3. In *Proceedings of the* IFIP *TC6/WG6.1* 14th *International Conference on Testing of Communicating Systems, (TestCom 2002), Berlin, Germany*, edited by I. Schieferdecker, H. König, & A. Wolisz, pages 407–424. The International Federation for Information Processing, IFIP, Kluwer Academic Publishers. ISBN 0-7923-7695-1.

Dai, Z., J. Grabowski, & H. Neukirchen, 2003. *TIMED*TTCN-3 Based Graphical Real-Time Test Specification. In *Proceedings of the* IFIP *TC6/WG6.1* 15th *International Conference on Testing of Communicating Systems, (TestCom 2003), Sophia-Antipolis, France*, edited by D. Hogrefe & A. Wiles, volume 2644 of *Lecture Notes in Computer Science, (*LNCS*)*, pages 110–127. The International Federation for Information Processing, IFIP, Springer Verlag. ISBN 3-540-40123-7. ISSN 0302-9743.

Dai, Z. R., J. Grabowski, H. Neukirchen, & H. Pals, 2004. From Design to Test with UML – Applied to a Roaming Algorithm for Bluetooth Devices. In *Proceedings of the* IFIP *TC6/WG6.1* 16th *International Conference on Testing of Communicating Systems, (TestCom 2004), Oxford, United Kingdom*, Lecture Notes in Computer Science,

(LNCS). The International Federation for Information Processing, IFIP, Springer Verlag.

Ebner, M., 2001a. A Mapping of OMG IDL to TTCN-3. SIIM Technical Report SIIM-TR-A-01-11, Institute for Telematics, University of Lübeck, Germany. Schriftenreihe der Institute für Informatik\Mathematik, (SIIM).

Ebner, M., 2001b. Mapping CORBA IDL to TTCN-3 based on IDL to TTCN-2 mappings. In *Proceedings of the* 11th GI/ITG *Technical Meeting on Formal Description Techniques for Distributed Systems, Bruchsal, Germany, 21.-22. June 2001*. International University in Germany. URL http://www.i-u.de/fbt2001/.

Ebner, M., 2004. TTCN-3 Test Case Generation from Message Sequence Charts. In ISSRE04 *Workshop on Integrated-reliability with Telecommunications and* UML *Languages (*ISSRE04:WITUL*), 2. November 2004,* IRISA *Rennes, France*. The Institute of Electrical and Electronics Engineers, IEEE. The 15th IEEE International Symposium on Software Reliability Engineering, (accepted paper, to be appear).

Ebner, M., A. Yin, & M. Li, 2002. Definition and Utilisation of OMG IDL to TTCN-3 Mappings. In *Proceedings of the* IFIP *TC6/WG6.1* 14th*International Conference on Testing of Communicating Systems, (TestCom 2002), Berlin, Germany*, edited by I. Schieferdecker, H. König, & A. Wolisz, pages 443–458. The International Federation for Information Processing, IFIP, Kluwer Academic Publishers. ISBN 0-7923-7695-1.

Ellsberger, J., D. Hogrefe, & A. Sarma, 1997. *SDL — Formal Object-oriented Language for Communicating Systems*. Prentice Hall Europe. ISBN 0-13-621384-7.

ETSI, 2001. Methods for Testing and Specification (MTS) — The Testing and Test Control Notation version 3 — Part 2: TTCN-3 Tabular Presentation Format (TPF). European Standard ETSI ES 201 873-2 v.2.2.0, ETSI, European Telecommunications Standards Institute, Sophia-Antipolis, France.

ETSI, 2002a. Methods for Testing and Specification (MTS) — The Testing and Test Control Notation version 3 — Part 1: TTCN-3 Core Language. European Standard ETSI ES 201 873-1 v2.2.1, ETSI, European Telecommunications Standards Institute, Sophia-Antipolis, France.

ETSI, 2002b. Methods for Testing and Specification (MTS) — The Testing and Test Control Notation version 3 — Part 3: TTCN-3 Graphical Presentation Format (GFT). Technical Report ETSI TR 101 873-3 v.1.1.2, ETSI, European Telecommunications Standards Institute, Sophia-Antipolis, France.

ETSI, 2003. Methods for Testing and Specification (MTS) — The IDL to TTCN-3 Mapping. Technical Specification ETSI TS 102 219 v1.1.1, ETSI, European Telecommunications Standards Institute, Sophia-Antipolis, France.

Grabowski, J., 2002. *Specification Based Testing of Real-Time Distributed Systems*. habilitation thesis, Universität zu Lübeck, Germany.

Grabowski, J., B. Koch, M. Schmitt, & D. Hogrefe, 1999. SDL and MSC Based Test Generation for Distributed Test Architectures. In *SDL '99 The next Millenium – Proceedings of the Nineth SDL Forum*, edited by D. R, G. v. Bochmann, & Y. Lahav. Elsevier, Montreal, Canada.

Grabowski, J., R. Scheurer, Z. Dai, & D. Hogrefe, 1997. Applying SAMSTAG to the B-ISDN protocol SSCOP. In *Proceedings of the* IFIP *TC6/WG6.1* 10th *International Workshop on Testing of Communicating Systems, (*IWTCS *1997), Cheju Island, Korea*, edited by M. Kim, S. Kang, & K. Hong. The International Federation for Information Processing, IFIP, Chapman & Hall.

Hogrefe, D., 1989. *Estelle, LOTOS und SDL. Standard-Spezifikationssprachen für verteilte System*. Springer Verlag. ISBN 3-540-50477-X.

ISO/IEC, 1990. Information Technology — ISO 7-bit coded character set for information exchange. International Standard 646, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ISO/IEC, 1993. Information Technology — Universal Multiple Octet-Coded Character Set (UCS). International Standard 10646, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ISO/IEC, 1994. Information Technology — Open Systems Interconnection — Conformance Testing Methodology and Framework — Part 1: General Concepts. International Standard 9646-1, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ISO/IEC, 1998a. Information Technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1. International Standard 8859-1, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ISO/IEC, 1998b. Information Technology — Open Systems Interconnection — Conformance Testing Methodology and Framework — Part 3: The Tree and Tabular Combined Notation (Second Edition). International Standard 9646-3, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ISO/IEC, 1998c. Information Technology — Programming Languages — C++. International Standard 14882, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ISO/IEC, 1999. Information Technology — Open Distributed Processing — Interface Definition Language. International Standard DIS 14750, ISO, International Organisation for Standardisation and IEC, International Electrotechnical Commission.

ITU-T, 1996. Recommendation: Message Sequence Chart (MSC). International Standard Z.120 (10/96), ITU-T, International Telecommunication Union — Telecommunication Standardisation Sector SG 10.

ITU-T, 1997a. Recommendation: Information Technology — Open Distributed Processing — Interface Definition Language (IDL). International Standard X.920, ITU-T, International Telecommunication Union — Telecommunication Standardisation Sector.

ITU-T, 1997b. Recommendation: Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. International Standard X.680, ITU-T, International Telecommunication Union — Telecommunication Standardisation Sector.

ITU-T, 1997c. Recommendation: Information Technology – Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. International Standard X.680, ITU-T, International Telecommunication Union — Telecommunication Standardisation Sector.

ITU-T, 1999. Recommendation: Specification and Description Language (SDL). International Standard Z.100 (11/99), ITU-T, International Telecommunication Union — Telecommunication Standardisation Sector SG 10.

ITU-T, 2001. Recommendation: Message Sequence Chart (MSC). International Standard Z.120 (11/99) with Corrigendum 1, ITU-T, International Telecommunication Union — Telecommunication Standardisation Sector SG 10.

Jeckle, M., C. Rupp, J. Hahn, B. Zengler, & S. Queins, 2004. *UML 2 glasklar*. Hanser, 1 edition. ISBN 3-446-22575-7.

Kaner, C., J. Falk, & H. Nguyen, 1999. *Testing Computer Software*. Wiley Computer Publishing, 2 edition. ISBN 0-471-35846-0.

Kerbrat, A., T. Jéron, & R. Groz, 1999. Automated test generation from SDL specifications. In *SDL '99 The Next Millenium, Proceedings of the Ninth SDL Forum, Montréal, Québec, Canada, 21–25 Hune, 1999*, edited by R. Dssouli, G. v. Bochmann, & Y. Lahav, pages 135–151. Elsevier.

Koch, B., 2001. *Test-purpose-based Test Generation for Distributed Test Architectures*. doctoral thesis, Universität zu Lübeck, Germany.

Koch, B., J. Grabowski, D. Hogrefe, & M. Schmitt, 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.

Kung, D., P. Hsia, & J. Gao (editors), 1998. *Testing Object-Oriented Software*. IEEE Computer Society. ISBN 0-8186-8520-4.

Li, M., 1998. *Testing Computational Interfaces of CORBA Service using TTCN and CORBA*. Diplomarbeit, Fachgebiet Telekommunikationsnetze, Fachbereich Elektrotechnik, Technische Universität Berlin, Germany.

Li, M., I. Schieferdecker, & A. Rennoch, 1999. Testing the TINA Retailer Reference Points. In 4^{th} *Int. Symposium on Autonomous Decentralized Systems (ISADS'99), Tokyo, Japan, Mar. 1999*.

McGregor, J. & D. Sykes, 2001. *Practical Guide to Testing Object-Oriented Software*. Object Technology Series. Addison-Wesley. ISBN 0-201-32564-0.

Mednonogov, A., 2000. Calypso Gateway specification, version 0.07. Technical report, Telecommunications Software and Multimedia Laboratory, Helsinki University of Technology, Finland.

Mednonogov, A., H. Kari, O. Martikainen, & J. Malinen, 2000. Conformance Testing of CORBA Services using Tree and Tabular Combined Notation. In *Proceedings of the* IFIP *TC6/WG6.1* 13th *International Conference on Testing of Communicating Systems, (TestCom 2000), August 29.–September 1., 2000, Ottawa, Canada*, edited by H. Ural, R. Probert, & G. Bochmann, pages 193–208. The International Federation for Information Processing, IFIP, Kluwer Academic Publishers.

Mellor, S. J. & M. J. Balcer, 2002. *Executable UML: A Foundation for Model-Driven Architecture*. Object Technology Series. Addison-Wesley. ISBN 0-201-74804-5.

Miga, A., D. Amyot, F. Bordeleau, C. Cameron, & M. Woodside, 2001. Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In *Proceedings of the SDL 2001: Meeting UML*. 10th *International SDL Forum Copenhagen, Denmark, June 27–29, 2001*, edited by R. Reed & J. Reed, number 2078 in Lecture Notes in Computer Science, (LNCS), pages 268–287. SDL Forum Society, Springer Verlag. ISBN 3-540-42281-1.

Mulvihill, B., 2003. Generation of TTCN Test Cases from Use Case Map Scenarios. Graduate student report, School of Information Technology and Engineering, University of Ottawa, Canada. Supervisor: Daniel Amyot, URL http://www.site.uottawa.ca/~damyot/students/BryanMulvihillRep.pdf.

Myers, G., 2001. *Methodisches Testen von Programmen*. Oldenbourg, 7 edition. ISBN 3-486-25634-3. (original title: The Art of Software Testing).

Neukirchen, H., 2004. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. doctoral thesis, Georg-August-Universität Göttingen, Germany.

OMG, 1997. CORBAservices - Naming Service. Technical Report formal/97-12-10, OMG, Object Management Group.

OMG, 1999. C Language Mapping Specification. OMG Formal Document formal/99-07-35, OMG, Object Management Group.

OMG, 2001a. Model Driven Architecture. Technical report, OMG, Object Management Group.

OMG, 2001b. The Common Object Request Broker — Architecture and Specification. OMG Formal Document formal/01-02-01, OMG, Object Management Group. Version 2.4.2.

OMG, 2003a. CORBA to WSDL/SOAP Interworking Specification. OMG Formal Document formal/03-11-02, OMG, Object Management Group. Version 1.0.

OMG, 2003b. Unified Modeling Language 2.0: Superstructure. OMG Final Adopted Specification ptc/03-08-02, OMG, Object Management Group.

OMG, 2003c. Unified Modeling Language Specification. OMG Formal Document formal/03-03-01, OMG, Object Management Group. Version 1.5.

OMG, 2003d. WSDL-SOAP to CORBA Interworking. OMG Draft Adopted Specification ptc/03-07-04, OMG, Object Management Group.

OMG, 2003e. XML Metadata Interchange (XMI) Specification. OMG Formal Document formal/03-05-02, OMG, Object Management Group. Version 2.0.

OMG, 2003f. XML/Valuetype Language Mapping. OMG Formal Document formal/03-04-01, OMG, Object Management Group. Version 1.1.

Open Group, 2000. Inter-Domain Management: Specification & Interaction Translation. Technical Standard C802, Open Group. URL http://www.jidm.org.

Savoye, R., 2001. *DejaGNU — The GNU Testing Framework*. Free Software Foundation, 1.4.1 revision 0.6.1 edition.

Schieferdecker, I., Z. Dai, J. Grabowski, & A. Rennoch, 2003. The UML 2.0 Testing Profile and its Relation to TTCN-3. In *Proceedings of the* IFIP *TC6/WG6.1* 15th *International Conference on Testing of Communicating Systems, (TestCom 2003), Sophia-Antipolis, France*, edited by D. Hogrefe & A. Wiles, volume 2644 of *Lecture Notes in Computer Science, (*LNCS*)*, pages 79–94. The International Federation for Information Processing, IFIP, Springer Verlag. ISBN 3-540-40123-7. ISSN 0302-9743.

Schieferdecker, I. & J. Grabowski, 2003. The Graphical Format of TTCN-3 in the Context of MSC and UML. In *Proceedings of the 3th International Workshop on SDL and MSC (SAM 2002), Telecommunications and beyond: The Broader Applicability of SDL and MSC, Aberystwyth, UK, June 24.-26., 2002. Revised Papers*, edited by E. Sherratt, volume 2599 of *Lecture Notes in Computer Science, (LNCS)*, pages 233–252. Springer Verlag. ISBN 3-540-00877-2. ISSN 0302-9743.

Schieferdecker, I., M. Li, & A. Hoffmann, 1998. Conformance Testing of TINA Service Components — The TTCN/CORBA Gateway. In *Proceedings of the 5th International Conference on Intelligence and Services in Networks, IS&N'98, Antwerp, Belgium, May 25–28, 1998*, edited by S. Trigila, A. Mullery, M. Campolargo, H. Vanderstraeten, & M. Mampaey, volume 1430 of *Lecture Notes in Computer Science, (LNCS)*, pages 393–408. Springer Verlag. ISBN 3-540-64598-5.

Schieferdecker, I. & B. Stepien, 2003. Automated Testing of XML/SOAP based Web Services. In 13th *Fachkonferenz der Gesellschaft für Informatik (GI) Fachgruppe "Kommunikation in verteilten Systemen" (KiVS), Leipzig, 26.–28. Febr., 2003*, edited by K. Irmscher & K. Fähnrich. Springer Verlag. ISBN 3-540-00365-7.

Schmitt, M., 2003. *Automatic Test Generation Based on Formal Specifications — Practical Procedures for Efficient State Space Exploration and Improved Representation of Test Cases*. doctoral thesis, Georg-August-Universität Göttingen, Germany. URL http://webdoc.sub.gwdg.de/diss/2003/schmitt/schmitt.pdf.

Schmitt, M. & M. Ebner, 2003. The TTCN-3 module and template concepts revisited. Technical Report IFI-TB-2003-02, Institut für Informatik, Georg-August-Universität Göttingen, Germany. ISSN 1611-1044.

Schmitt, M., M. Ebner, & J. Grabowski, 2000. Test Generation with Autolink and TestComposer. In *Proceedings of the 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM'2000), Grenoble, France, June 26.–28., 2000*, pages 218–232.

Yin, A., 2001. *Testing Operation-Based Interfaces — Exemplified for CORBA with ADL and TTCN-3*. Diplomarbeit, Telecommunication Network Group, Faculty of Electrical Engineering and Computer Science, Technical University Berlin, Germany.

Yin, A., I. Schieferdecker, & M. Li, 2001. Mapping of IDL to TTCN-3. Technical report, Fraunhofer Institute for Open Communication Systems (FOKUS), Germany.

List of Figures

1.1. Fundamental concept of the thesis 2

2.1. The V process model for software development (Balzert 1998, page 101) 7
2.2. Test generation with TestComposer and Autolink . . . 17
2.3. User's view of TTCN-3 core language, presentation formats, and imported types 20
2.4. The Inres service and protocol 22
2.5. The local test method applied to Inres 23
2.6. Message- and blocking procedure-based communication . 28
2.7. Conceptual view of a typical TTCN-3 test configuration . 32

3.1. UML-based test specification 44
3.2. MSC document example 46
3.3. Basic MSCs for the Inres protocol 47
3.4. HMSC example 52
3.5. The CORBA architecture 54

4.1. Basic MSC to TTCN-3 mapping concept 62
4.2. Basic instance structure to specify test cases via MSC . . . 63
4.3. MSC to TTCN-3 base mapping 67
4.4. MSC to TTCN-3 alternative mapping 71
4.5. MSC to TTCN-3 optional mapping 72
4.6. MSC to TTCN-3 exception mapping 73
4.7. MSC to TTCN-3 loop mapping 1 73
4.8. MSC to TTCN-3 loop mapping 2 74
4.9. MSC to TTCN-3 mapping of HMSCs 75

List of Tables

2.1. Overview of TTCN-3 types 25
2.2. Overview of TTCN-3 type variants respectively *useful* types 26

3.1. Overview of IDL types 56

4.1. Conceptual list of MSC to TTCN-3 mapping 77

5.1. IDL to TTCN-3 literal mapping 81
5.2. IDL and TTCN-3 operators for constant expressions . . 86
5.3. IDL to TTCN-3 mapping for basic types 91
5.4. IDL to TTCN-3 mapping for constructed types 93
5.5. IDL to TTCN-3 mapping for template types 95
5.6. IDL to TTCN-3 mapping for complex types 95
5.7. IDL to TTCN-3 mapping for interface elements 102

6.1. Corresponding data types for operations in TTCN-3 . . . 112
6.2. Class element accessibility by access attributes 115

A.1. Conceptual list of IDL mapping 121
A.2. Comparison of IDL, ASN.1, TTCN-2, and TTCN-3 data types 123