SDL and MSC Based Test Case Generation – An Overall View of the SAMSTAG Method

Jens Grabowski

IAM-94-005

May 1994

Abstract

This technical report summarizes the results of the research and development project *'Conformance Testing – A Tool for the Generation of Test Cases'*.[1] Within this project we developed a method for the automatic generation of test cases based on formal specifications and formally defined test purposes. The method is called SaMsTaG. It is implemented in the SaMsTaG tool. Most of the work has already been published in conference proceedings [13, 30], technical reports [12, 14, 15] and project reports [11, 5, 6, 7, 8, 9, 10]. For detailed information these publications should be consulted.

The report starts with a short introduction (Section 1). Then the standardized conformance testing procedure [22], in the following abbreviated by CTMF/FMCT, is compared with other test case generation methods (Section 2). Afterwards, the SaMsTaG method (Sections 4, 5) and the SaMsTaG tool are introduced (Section 6). In the last section the formal aspects of CTMF/FMCT, other test case generation methods and the SaMsTaG method are summarized (Section 7). This summary provides a possibility for a complete formal explanation of the entire conformance testing procedure.

CR Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General; C.2.2 [Computer-Communication Networks]: Network Protocols; D.2.5 [Software Engineering:] Testing and Debugging

General Terms: Validation, Test Case Generation, Test Case Specification

[1]The project was funded by Swiss PTT under the contract no. 233/257.

1 Introduction

For approximately 10 years the ISO/IEC and the ITU-TS[2] have been working on methods for protocol conformance testing. A result of these investigations is the international ISO/IEC standard 9646[3] *'OSI Conformance Testing Methodology and Framework'* (CTMF) [22]. CTMF consists of seven parts. It includes general concepts for conformance testing, several test methods, the *'Tree and Tabular Combined Notation'* (TTCN) [24] as description language for test cases, and information concerning test realization, requirements on test laboratories, protocol profile test specification, and implementation conformance statements. CTMF also seems to be a good basis for industrial protocol testing. The growing availability of commercial tools which support and automate several parts of standardized conformance testing procedure emphasizes the broad acceptance and increasing dissemination of CTMF.

In parallel with the development of CTMF, ISO/IEC and ITU-TS developed the formal description techniques SDL, Estelle, LOTOS [16] and MSC [26]. The application of these languages in protocol standards should support implementation, validation and conformance testing of communication protocols. For this reason the joint project *'Formal Methods in Conformance Testing'* (FMCT) [18, 23] of ISO/IEC and ITU-TS formalizes CTMF, i.e. adapts and extends CTMF to meet the specific needs of conformance testing for formally specified protocols.

This chapter also deals with the conformance testing of formally described protocols. Therefore CTMF and FMCT can not be treated independently from each other. We refer to the work of both, i.e. to [22] and [23], by means of the abbreviation CTMF/FMCT.

The use of formal methods in protocol specification also resulted in scientific research in the area of conformance testing. A result of this work was the development of several methods for the automatic generation of test cases. In the following we refer to these methods with the term *scientific methods*. A discussion of such methods for example can be found in [19] or [27].

All scientific methods try to prove a relation, the so-called *conformance relation*, between a specification and an implementation by means of a test. The test case generation is determined by the specification and the used conformance relation. The scientific methods have to deal with three main problems. The first problem concerns the conformance relation. Most of the used conformance relations can be used only for protocol specifications which can be described by finite automata. In practice, this restriction is hardly ever met. The second problem concerns the complexity of real protocols. Even if the conformance relation can be checked, often the number of necessary tests is so large that they are not manageable within an industrial environment. The third problem concerns the relation to CTMF/FMCT. Most concepts and procedures of the scientific methods can not be related directly to the terms and procedures defined in CTMF/FMCT.

However, there exists a possibility to bring CTMF/FMCT and the scientific methods closer together. For this, the results of the research project *'Conformance Testing - A Tool for the Generation of Test Cases'* may play an important role. The project was closely

[2]In March 1993 the CCITT became the *'Telecommunications Standards Sector of the International Telecommunication Union'* (ITU-TS).

[3]ITU-TS issues CTMF as recommendation X.290.

aligned with CTMF/FMCT. A main goal of the project was to formalize an important step of the standardized procedure for conformance testing and to automate it by means of a prototype tool. The selected step concerns the generation of abstract test cases based on a formal protocol specification and a set of test purposes. Since in CTMF/FMCT the term *test purpose* is not defined formally, we formalized it and developed a method for the automatic generation of test cases. The method is called SaMsTaG (Sdl And Msc baSed Test cAse Generation) method and is implemented in the SaMsTaG tool.

2 CTMF/FMCT and scientific methods

This section starts with a description of the conformance testing procedure according to CTMF/FMCT. Then, the scientific methods are explained and afterwards, the similarities and differences of both procedures are summarized.

2.1 Conformance testing according to CTMF/FMCT

Figure 1 presents the CTMF/FMCT conformance testing procedure to derive a conformance statement by means of a test which is based on a protocol specification and a protocol implementation.[4] The rectangles denote actions and the ellipses describe the inputs and outputs of the actions. Dashed rectangles and ellipses represent actions, inputs, and outputs which are described informally in CTMF/FMCT. Letters and numbers serve as references for the following description.

The goal of conformance testing is to prove that a protocol implementation (a) has the behavior which is described in a protocol specification (b). The implementation is not verified directly against the specification, but tested with a set of test cases, a so-called *test suite*.

For deriving a test suite from a protocol specification a set of *test purposes* (c) has to be defined. A test purpose is an informal description of a behavior or a property which shall proven by the conformance test. Based on the test purposes and the protocol specification an *abstract test suite* is developed. An abstract test suite consists of *abstract test cases*. In CTMF/FMCT the definition of the test purposes (1), the test purposes itself (c) and the specification of the abstract test suite (2) are described only informally. In practice the actions (1) and (2) are performed manually by experts.

An abstract test case describes the required exchange of protocol data units (PDUs) and service primitives (SPs) independent from the protocol implementation and test realization. In order to transform an abstract test suite (d) into a *executable test suite* (e), all PDUs and SPs have to be converted into bit combinations which can be interpreted by the test equipment, i.e. test processes or test devices.

During the *conformance test* (4) the implementation is stimulated by inputs and the resulting outputs are observed. The inputs and outputs are described within the individual

[4]CTMF/FMCT defines a very comprehensive and complex procedure. Therefore Figure 1 cannot describe all aspects of the entire conformance testing procedure. E.g. the influences of PICS and PIXIT are not considered.

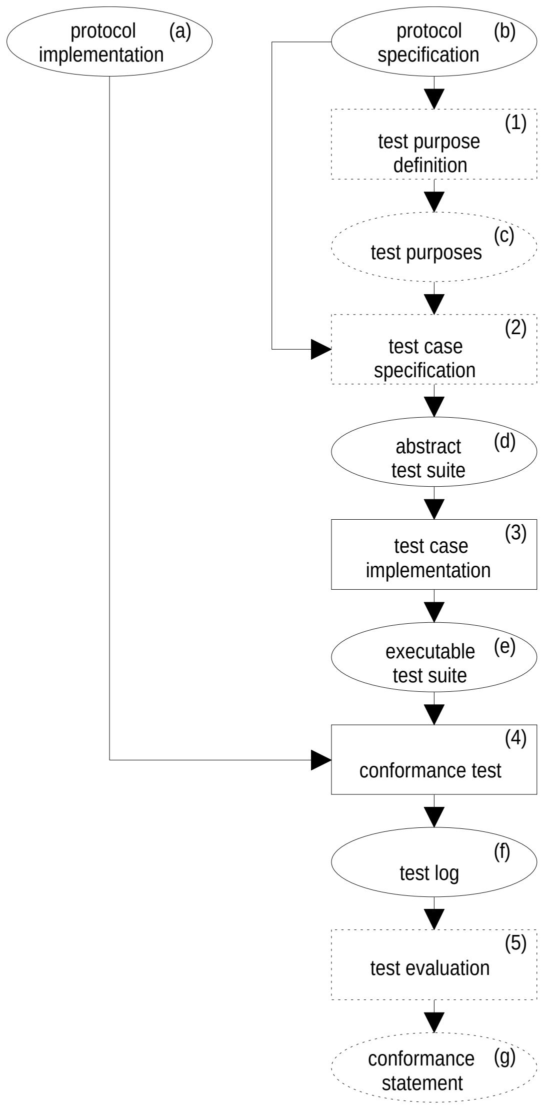

protocol implementation (a)
protocol specification (b)
(1) test purpose definition
test purposes (c)
(2) test case specification
abstract test suite (d)
(3) test case implementation
executable test suite (e)
(4) conformance test
test log (f)
(5) test evaluation
conformance statement (g)

Figure 1: The conformance testing procedure according to CTMF/FMCT

test cases. According to the shown behavior one of three possible test verdicts is assigned to each test case. The whole test campaign is recorded in a *test log* (f).

The *evaluation of the test* (5) and the following *conformance statement* are described only informally in CTMF/FMCT. One reason for this is that, besides the test log, a general conformance statement also should consider further technical and non technical aspects, e.g. the reputation of the manufacturer.

Only the actions (3) and (4) in Figure 1 can be automated. Currently, the actions (1), (2) and (5) are described informally. Therefore, they can not be automated. The input for the definition of the test purposes (1) is a protocol specification. We assume that it is written in a standardized formal description technique, i.e. LOTOS, Estelle, or SDL. Although the behavior of the protocol is unambiguously described by the protocol specification, action (1) can not be automated, because the central term *test purpose* and its relation to the protocol specification still is an open question. Since the set of

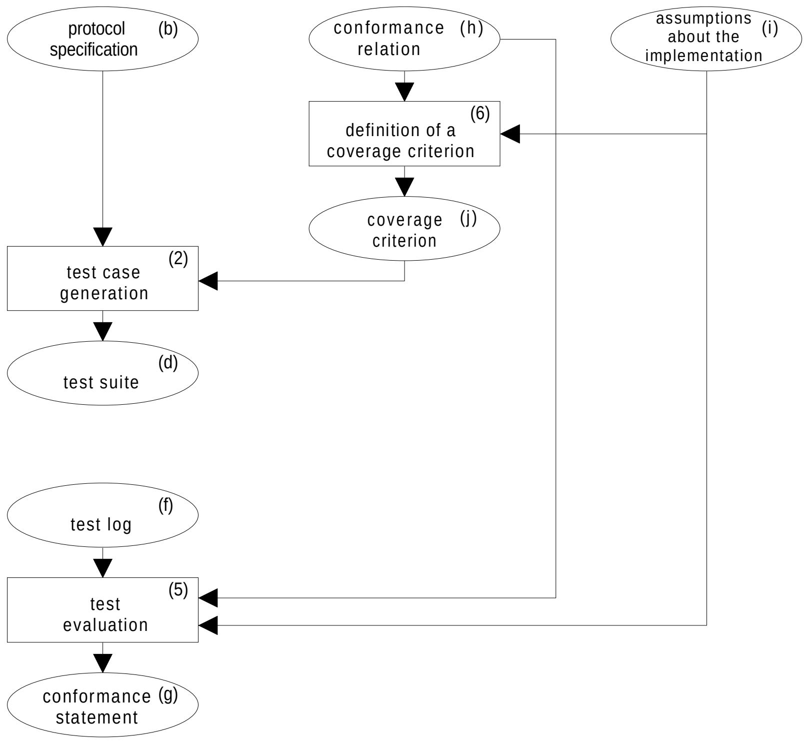

protocol specification (b)
conformance relation (h)
assumptions about the implementation (i)
definition of a coverage criterion (6)
coverage criterion (j)
test case generation (2)
test suite (d)
test log (f)
test evaluation (5)
conformance statement (g)

Figure 2: The general procedure of test case generation with scientific methods

test purposes is the prerequisite for the specification of the abstract test suite (2), this action also can not be automated. For the manual specification of the abstract test cases the standardized test case description language TTCN [24] can be used. Action (3) can be performed automatically by means of commercial TTCN editors, compilers and interpreters (e.g. [33]). Also the technical equipment for the automatic execution of the conformance tests is commercially available.

2.2 Test case generation with scientific methods

Methods for test case generation which are developed in a scientific environment, in the following called *scientific methods*, generate test cases from a formal specification in order to prove a *conformance relation* between specification and implementation.

In the following we explain the general procedure of scientific methods by means of two examples and Figure 2. Example A refers to methods which are discussed by Holzmann in [19]. They can be used to generate test cases for finite automata. In example B we discuss a method which can be applied to generate test cases for LOTOS specifications. It is described by Brinksma in [3]. The letters and numbers in the following text serve as references to the rectangles and ellipses in Figure 2.

In general, all scientific methods start with the definition of a *conformance relation* (h). This relation states something about the relation between the traces of the protocol specification and the protocol implementation. The validity of the relation should be verified by the conformance test.

Unfortunately, there exist conformance relations which can not be proven for arbitrary specifications and implementations. For example, within finite time a behavioral equivalence can only be checked for specifications and implementation which can be described by finite and deterministic automata. This means that the conformance relation restricts the set of testable specifications and implementations.

For conformance testing the *protocol specification* (b) is given and it can be checked directly whether the specification is testable for the chosen *conformance relation* (h). In general, the implementation is given as *black box*, i.e. conformance testing is *black box testing* and it it can not be checked if the implementation is testable or not. Therefore, one has to make *assumptions about the implementation* (i). Thus, the validity of the *conformance statement* (g) depends on the validity of the assumptions.

Approach A. The scientific methods which are discussed by Holzmann attempt to prove a behavioral equivalence between specification and implementation. It is assumed that the time available for testing is finite. To prove behavioral equivalence specification and implementation have to behave like finite, strongly connected and deterministic automata. In conformance testing the implementation is given as black box. Therefore it can not be proven that the implementation behaves like the required automaton. We only can make assumptions about the behavior of the implementation (cf. Figure 2 (i)).

Approach B. Brinksma has worked on testing behavioral equivalence in the realm of labeled transition systems, i.e. infinite automata.[5] His theory can handle labeled transition systems, because he does not assume that the time for testing is finite.

Based on the conformance relation (h) and the assumptions (i) a *coverage criterion* (j) is determined. This criterion defines how the specification should be covered by the test cases. For example, a coverage criterion may require that all state transitions have to be performed for at least once. Based on the coverage criterion and the protocol specification the *test cases* (d) are generated. The scientific methods have no notion about the term *test purpose* which is a central concept of CTMF/FMCT (cf. (c) in Figure 1).

Approach A. To test the behavioral equivalence of two finite automata all state transitions have to be checked. But, it is not necessary to develop an individual test case for each state transition. By means of a so-called *transition tour* only one big test case is generated. This test case checks all state transitions at once.

Approach B. The approach of Brinksma also tests the entire behavior of the specification, i.e. all state transitions. But, instead of producing test cases, he generates a so-called *canonical tester*. A canonical tester is a labeled transition system which describes the environment of the specification. It can be considered to be the inversion of the specification.

The conformance test itself is not treated by the scientific methods. But, it is assumed that there exist a *test log* (f) which, together with the assumptions (i) and the conformance relation (h), can be used for the *evaluation of the whole test* (5). The result of the evaluation is a *conformance statement* (g) which states whether the conformance relation is proven or not.

[5]Brinksma uses LOTOS [21] as specification language.

Approach A. The conformance relation is proven if the test case which is generated by the transition tour has been executed successfully. As stated earlier, the validity of this statement can only be guaranteed if the assumptions about the implementation are valid.

Approach B. For test execution the canonical tester is connected with the implementation which should be tested. The result is a closed system, i.e. all communication is system internal. If the system does not terminate, nothing can be said about the validity of the conformance relation. If the system ends with an allowed trace, i.e. a trace where canonical tester and specification end, the test is successful. If all allowed traces can be tested, the validity of the conformance relation is proven. The conformance relation does not hold when the system presents a trace that is not allowed.

2.3 A comparative summary

By starting with the test case development and ending with the conformance statement CTMF/FMCT covers the whole conformance testing procedure. But, the individual steps are defined with a different degree of formality. As a consequence, they also differ in the possibility to be automated.

The whole test execution (cf. Figure 1) which starts with the abstract test suite (d) and ends up with the test log (f) has already been automated. Therefore, it is sufficiently formally defined. In contrast to this, the terms test purpose (c), conformance statement (g) and the actions test purpose definition (1), test case specification (2) and test evaluation (5) are described only informally. This is problematic for several reasons.

The development of the abstract test suite is based on two informal steps which, in most cases, are performed manually by protocol experts. The goal of the expert is to produce a test suite which checks all functions of the protocol. How close this goal is reached heavily depends on the definition of the test purposes (1). Without formal criteria it is difficult to judge the set of test purposes.

For the test case specification (2) the protocol expert interprets the test purposes and transforms his interpretation into corresponding test cases. The test cases are written in a test case description language. The interpretation of test purposes and transformation into test cases are non trivial tasks, and therefore a test suite may include misinterpretations and errors. These problems may influence the evaluation of the whole conformance test (5). The concluding conformance statement may become very vague without any criterion which judges the quality of the test suite.

The scientific methods in [3] and [19] (cf. Figure 2) formalize the test case generation (2) and the test evaluation (5). This is done by means of a conformance relation which defines the goal of the test. The conformance relation determines a coverage criterion (j). Based on the coverage criterion and the protocol specification a test suite can be generated automatically. The test is evaluated (5) by checking whether all generated test cases have been performed successfully. In this case the conformance relation is proven, i.e. the goal of the test is reached.

The main problems of the scientific methods are the conformance relation and the complexity of real systems. There exist conformance relations which are not provable for arbitrary specifications and implementations. Therefore, a conformance relation restricts the set of testable specifications and implementations. Most conformance relations can not be proven for real protocols. Even if a conformance relation is theoretically testable, the number of necessary test runs often is so big that in practice they can not be performed.

Within the scientific methods the conformance relation and the corresponding coverage criterion determine the quality of the test suite. But, because of the mentioned problems the scientific methods can rarely be used for conformance testing. As a consequence, other criteria for the judgement of test suites and test logs have to be found. In industrial practice the test purposes play an important role.

A manufacturer has to prove that his product fulfills the requirements of a customer. For this, it might be necessary to give test cases and test logs to the customer. He can make a critical test review and then judge whether the requirements are fulfilled or not. But, such a test review only is possible if the purpose of each test case is known. On the other side, the test purposes may also be helpful for the manufacturer. Test purposes may help to locate implementation errors, if a test case fails.

3 Fundamental notions

For the explanation of the SAMSTAG method, the test case generation with the SAMSTAG method and the SAMSTAG tool we need to introduce some fundamental notions. Most of them are taken from the CTMF/FMCT work [22] and from the ITU-TS recommendations Z.100 [25] and Z.120 [26]. Only the terms *trace* and *observable* are new. We need them to simplify the description in the following sections.

3.1 SDL

The *'Specification and Description Language'* (SDL) is a standardized specification language for the behavior description of distributed systems. An SDL specification describes a set of extended finite state machines, so-called *processes*, which communicate asynchronously by means of messages. The entire behavior of an SDL specification can be described formally by a *labeled transition system*, i.e. by an infinite automaton. The states of the labeled transition system are defined by the global system states of the SDL specification. Such a global system state comprises the local states of the SDL processes, the variable values, and the not processed messages. The state transitions of the labeled transition system are defined by the state transitions of the SDL processes. In the following we do not need specific SDL features. We assume that the behavior of an SDL specification is given by a labeled transition system.

3.2 MSC

Also the *'Message Sequence Chart'* (MSC) language is a standardized language. MSC is a graphical trace language which admits a particularly intuitive representation of system runs in distributed systems. Formally, an MSC[6] defines a set of partially ordered message send and message receive actions. The behavior which is represented by an MSC can be described by a finite automaton. The automaton accepts all sequences of send and receive actions which are consistent with the partial order in the MSC. More detailed information can be found in [31].

3.3 IUT, SUT, tester and test architecture

IUT is an abbreviation for *'Implementation Under Test'*. It denotes a protocol implementation which should be tested.

In CTMF/FMCT it is assumed that the interface of an IUT to the next lower layer is not freely accessible. Therefore, this interface has to be controlled and observed via the service of the next lower layer, i.e. via other implementations. CTMF/FMCT uses the term *system under test* (SUT) to denote an IUT and the implementations which are necessary to interface the IUT.[7]

During the conformance test the SUT is driven and controlled by *testers*. A tester might be a software process, a test device, or a person which stimulates and observes the SUT manually.

A *test architecture* consists of the SUT and the test environment, i.e. all testers. CTMF/FMCT describes different test architectures for conformance tests. They mainly differ in the possibilities to access and control the IUT.

An example may clarify the different notions. Figure 3 presents a test architecture for the Inres protocol [17]. The IUT is the Initiator. The lower interface of the Initiator has to be tested via the Medium service. Therefore, the SUT consists of the Initiator and Medium service. The testers are the upper tester UT and the lower LT.

For the SAMSTAG tool it is required that a whole test architecture is described by an SDL specification. This offers the flexibility to generate test cases for different test architectures.

3.4 Trace and Observable

The behavior description of a specification (e.g. given in form of an SDL specification or an MSC) may include observable, and not observable (system internal) actions. Therefore, we distinguish between a *trace* and the *observable* of a trace. A trace is a sequence of arbitrary actions of a specification. Related to SDL, a trace may include arbitrary SDL actions, e.g. *input*, *output*, *task*, *decision*, *set*, or *reset*. In the following we use MSCs for the clear description of SDL traces. But, it should be noted that, because of its partial order representation, an MSC may describe a whole set of traces and observables.

The *observable* of a trace is the sequence of observable actions of a trace. An observable relevant for conformance testing may for example only include the actions of the testers. Figure 5 presents a trace of an SDL specification in form of an MSC and a corresponding

[6]In general, the term *MSC* is used for a diagram written in the MSC language and the language itself. Where necessary, we distinguish between both by using the terms *MSC language* and *MSC diagram*.

[7]For the test it is assumed that the implementations which are necessary to interface the IUT work correctly, i.e they do not influence the course of the test.

observable. The observable only includes the actions of LT and UT. It should be noted that there exists no unique relation between an observable and a trace. Different traces may have the same observable.

3.5 Test case

A test case consists of *preamble*, *test body* and *postamble*. Each part describes certain actions of the test processes (or devices). The preamble should drive the IUT from its initial state[8] into a state from which the test body can be performed. The test body defines the actions which should be executed in order to reach the test purpose, and the postamble drives the IUT back into its initial state. Furthermore, a complete test case should consider unforeseen responses of the IUT.

We define a test case as a set of observables. The observables describe action sequences of the testers. Each observable leads to a unique test verdict.

3.6 Test verdicts

The possible test verdicts are *pass*, *inconclusive* and *fail*. *Pass* is given if the test purpose is reached and if the test run ends in the initial state of the tested protocol.[9] A *fail* is assigned if a response of the IUT is not allowed by the specification. *Inconclusive* is given if neither a *pass* nor a *fail* can be assigned

3.7 TTCN

The *'Tree and Tabular Combined Notation'* (TTCN) is a standardized test case description language [24] which should be used for the specification of abstract test suites. The Figures 8 and 9 present examples for TTCN descriptions.

In a TTCN table the observables of a test case are described by means of a tree notation (cf. column *'Behaviour Description'* in Figure 8). The tree structure is determined by the order and the indentation of the specified actions. In general, the same indentation denotes a branching, i.e. alternative actions (e.g. lines Nr. 8 and Nr. 13), and the next larger indentation describes a subsequent action (e.g. lines Nr. 1 and Nr. 2).

Actions are characterized by the involved instance (i.e. LT and UT), by its kind (i.e. '!' denotes a send action, '?' describes a receive action) and by the message which has to be send or received. An example may clarify the notation. The statement 'UT!ICONreq' describes the sending of ICONreq to the IUT via the UT. TTCN allows to specify actions with arbitrary messages by using the OTHERWISE statement (e.g. UT?OTHERWISE in Figure 9).

Test verdicts are defined within a verdict column of a TTCN table. The verdict column of Figure 8 only includes *pass* and *inconclusive* verdicts. In this example *fail* behavior is specified by a *default behavior description* which is shown in Figure 9. Such defaults have

[8]CTMF/FMCT requires a stable testing state as the start and end state of the test case. This state might be an initial or and idle state.

[9]CTMF/FMCT allows several alternatives for the assignment of a *pass* verdict. Another possibility is to give a *pass* if the test purpose is reached, although the test run does not lead back to the initial state.

to be referenced in the test case header (cf. *Default* in Figure 8). TTCN offers much more facilities like *Constraints*, *Labels*, or *Timer* which are not relevant for the understanding of this chapter. A tutorial on TTCN can be found in [28].

4 The theory of the SAMSTAG method

In the previous sections it is shown why the scientific methods in [3] and [19] only partially can explain the standardized CTMF/FMCT conformance testing procedure [22, 23]. Especially, scientific methods have no notion of *test purposes* which are central for CTMF/FMCT. Contrary to the scientific methods, during the development of the SAMSTAG method we were guided by the CTMF/FMCT work. As a consequence, we formalized the term *test purpose*.

The SAMSTAG method is developed within the research and development project *'Conformance Testing a Tool for the Generation of Test Cases'* which is funded by the Swiss PTT. The goal of this project is the development of a method and a tool which allows to generate TTCN test cases [24] based on protocol specifications written in SDL [1, 25] and Message Sequence Charts (MSCs) [4, 26]. It is assumed that the allowed behavior of the protocol which should be tested is defined by an SDL specification and that the purpose of a test case is given by an MSC. The SAMSTAG method can be related to the general CTMF/FMCT procedure of conformance testing (cf. Section 2.1). In Figure 1 the *test case specification*, i.e. Action (2), is formalized.

SAMSTAG is an abbreviation of *'Sdl And Msc baSed Test cAse Generation'*. The abbreviation reflects the original project goal. But, we generalized the method in such a way, that it also can be used for protocols and test purposes which are not given by SDL specifications and MSCs.

The SAMSTAG method formalizes test purposes and defines the relation between test purposes, protocol specifications and test cases. Furthermore, it includes the algorithms for the test case generation. In this section we explain the meaning of test purposes and describe the relation. The algorithms are described in the next section.

4.1 A property oriented view on testing

Specifications and implementations can be looked at as generators for traces. They define finite or even infinite sets of traces and observables. We define a set of traces to be a *property*. An implementation has the property of a specification if its set of traces is a subset of the trace set of the specification.

There exist several classes of properties. Manna and Pnueli [29] distinguish between *guarantee properties*, *safety properties* and four higher classes of properties. Informally said, a guarantee property states that in each of its traces something *good* happens.[10] In contrast to this, a safety property states that in each of its traces never something *bad* happens. The higher properties state that in each trace always something *good* happens, or that from a certain point of time something *good* continuously happens.

[10]Therefore a guarantee property also can be interpreted as a *reachability criterion*.

The objective of testing means to compare the traces of an implementation and a specification. By this we try to find out something about the relation of the two sets of traces. In principle, we make statements about the properties of the specification which are shared or are not shared with the implementation. We call the classes of properties for which we can make these statements *testable properties*.

In [32] we only identified *guarantee* and *safety properties* to be testable properties. Informally said, a safety property is testable because during the test an implementation is able to show something *bad*. In this case it is proven that the implementation does not have the safety property. A guarantee property is testable because during the test an implementation is able to show the *good* thing. In this case the guarantee property is validated. Higher properties can neither be validated nor be violated. A finite test can not check whether something *good* happens arbitrarily often.

4.2 Safety and guarantee properties in conformance testing

Safety and guarantee properties can also be found in the conformance testing procedure according to CTMF/FMCT. The allowed system behavior is defined by a specification. Therefore the specification can be interpreted as a safety property.[11] In contrast to this a test purpose defines something which during the test should be observed. Thus, a test purpose can be interpreted as a guarantee property.

A test case is determined by a safety property, i.e. a system specification, and a guarantee property, i.e. a test purpose. The test verdicts *pass*, *fail* , and *inconclusive* follow directly from the combination of statements which during the test can be made about the two properties.

- *pass* is assigned to an observable which proves uniquely the guarantee property and which does not violate the safety property.
- *inconclusive* is assigned to an observable which does not prove the guarantee property, but also does not violate the safety property.
- *fail* is assigned to an observable which violates the safety property, regardless whether the guarantee property is proven or not.[12]

4.3 The representation of safety and guarantee properties

For the automatic generation of test cases a formal representation of safety and guarantee properties is needed. Possible formalisms for example are Petri nets, automata models, or temporal logic formulas. For the SAMSTAG method we choose an automaton model.[13]

[11]In general, a specification also may describe other properties. If temporal logic formulas are used as specification language, it is for example possible to specify liveness properties. For conformance testing we are only interested in the fact that a specification can be interpreted as guarantee property.

[12]The *fail* cases include the situation where the guarantee property is proven and the safety property is violated, although the situation should never appear. It makes no sense to design a test purpose which is not allowed by the specification.

[13]A short discussion about the advantages and disadvantages of the different models can be found in [12].

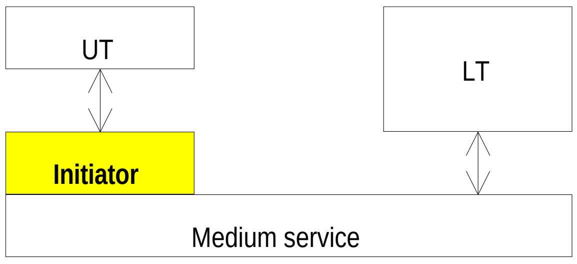


Figure 3: Test architecture for the Initiator entity of the Inres protocol

The SAMSTAG method assumes that a safety property is given by a labeled transition system, i.e. by an infinite automaton. The labeled transition system accepts all traces which do not violate the safety property. This approach is general enough to handle safety properties which are specified with a standardized formal description technique, i.e. LOTOS, Estelle, or SDL [16]. A guarantee property is represented by a finite automaton. The automaton accepts all traces which validate the property. Examples for formalisms which can be used to specify guarantee properties are the MSC language and temporal logic formulas [34].

5 Test case generation by SAMSTAG

In general, the automatic generation of test cases is based on the simulation of a test architecture. The simulation is driven by the test purpose and the test case definition (cf. Section 3.5). The observables which build the basis for the test case description are gained by recording the actions of the testers. In this section we describe the test case generation with the SAMSTAG method by means of a small example. The example is based on the Inres protocol [17].

Figure 3 presents a test architecture. We assume that it is given in form of an SDL specification. The Initiator entity of the Inres protocol should be tested. The test purpose is the MSC in Figure 4. It describes a special situation of the connection establishment phase.

The Initiator receives a connection request ICONreq from the upper tester UT and then sends a CR to a remote entity which in our case is simulated by the lower tester LT.[14] Afterwards, it waits for a connection confirmation CC. If after a certain time limit no CC is received, the Initiator is able to retransmit the CR for three times. In our case the lower tester LT answers after the observation of the third CR. The Initiator indicates the reception of the CC by sending an ICONind to the upper tester UT.

A test case consists of a finite set of observables. One of three test verdicts is assigned to each observable. Therefore we distinguish between *pass*, *fail* and *inconclusive observables*. The algorithms for the test case generation are based on the calculation of

[14]The lower interface of the Initiator entity has to be controlled via the Medium service. The protocol data units CR, CC, DT, or DR must be sent and received as parameters of the service primitives MDATreq and MDATind. The Medium service transmits the protocol data units transparently and has no influence on the behavior of the Initiator. Therefore in the following we abstract from the service primitives MDATreq and MDATind.

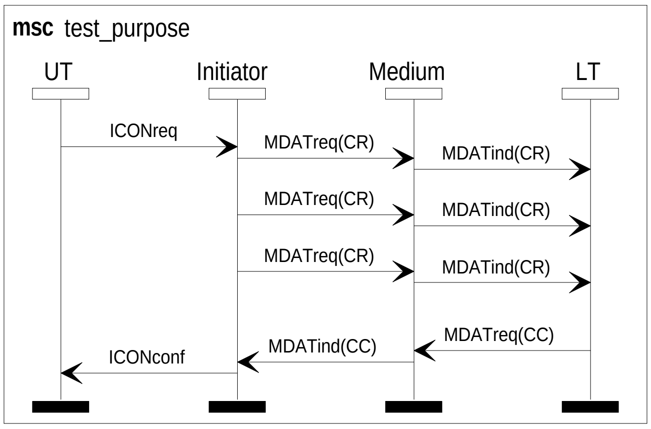


Figure 4: Connection establishment after the reception of the CR

these observables. The observables are generated in four steps which are reflected in the architecture of the SAMsTAG tool (cf. Figure 11).

1. In a first step so-called *possible pass observables* are calculated. A possible pass observable is the observable of a trace for which the following two conditions hold:

   (a) The trace starts and ends in the initial state of the SDL system.

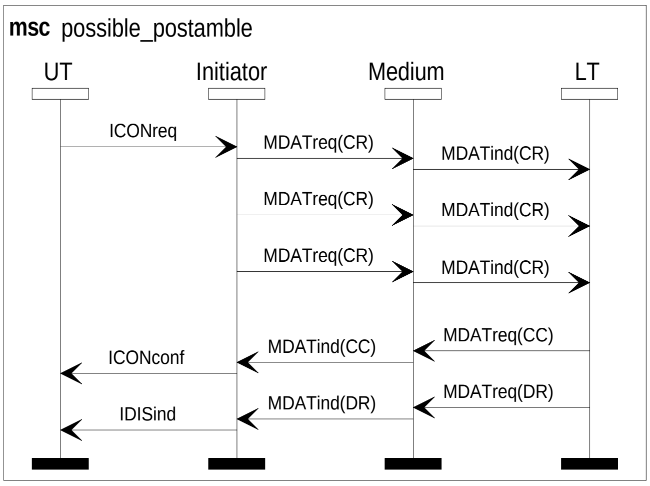
   (b) The trace includes the signal exchange which is specified in the MSC.

   The calculation of a possible pass observable starts with the search of a preamble which drives the SDL system into a state from which the signal exchange of the MSC can be observed. The preamble of our example is empty (cf. Figure 4). The MSC starts in the initial state. After the observation of the test purpose, i.e. the signal exchange of the MSC, the tested system has to be driven back into its initial state. A possible postamble is a normal disconnection (c.f. Figure 5). The lower tester LT initiates the disconnection by sending a disconnection request DR. On receipt of the CR the Initiator indicates the disconnection by sending an IDISind to the upper tester UT. The observable of the described trace is a possible pass observable.

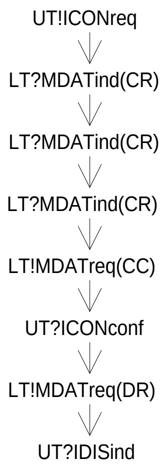
   In general, no unique relation exists between a trace and an observable. Different traces may have the same observable. The observable in Figure 6 (b) is identical with the observable in Figure 5 (b) with the except of the final event MDATind(CR).

   The SDL semantics allow no assumptions about the time which we have to wait for a possible MDATind(CR) after the reception of the IDISind. Therefore the observation of the IDISind does not guarantee the carrying out of the test purpose. If after the reception of the IDISind a fourth MDATind(CR) is observed, the test verdict *inconclusive* has to be assigned (cf. Figure 6).

2. In a second step the uniqueness of the obtained possible pass observables is proven, i.e. for each possible pass observable it is checked whether all traces which have the possible pass observable as observable fulfill the conditions (a) and (b) on page 13. In this case we call an observable *unique*, or *unique pass observable*. In general, there

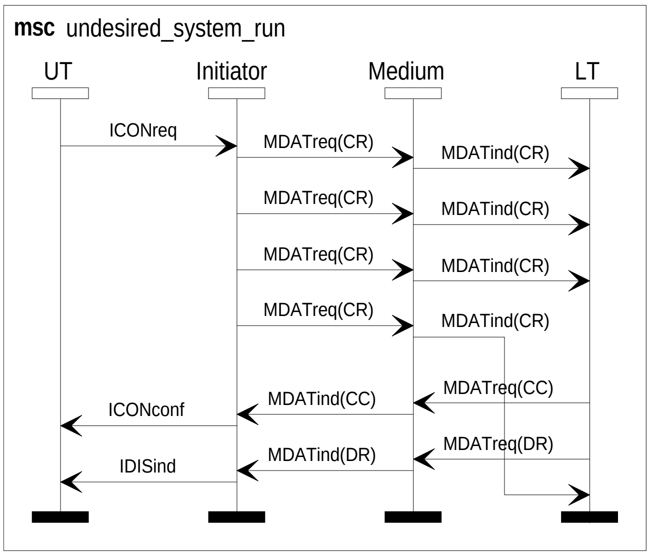


(a) MSC

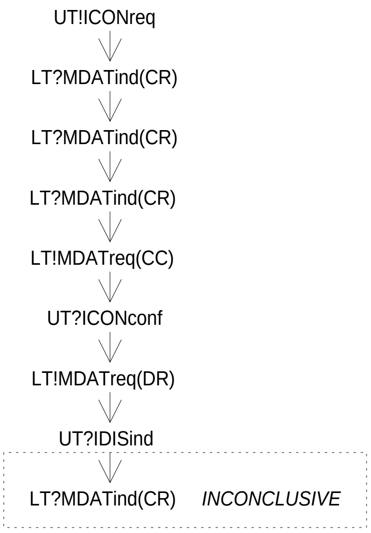


(b) Possible pass observable

Figure 5: MSC of Figure 4 with possible postamble and a corresponding observable



(a) MSC



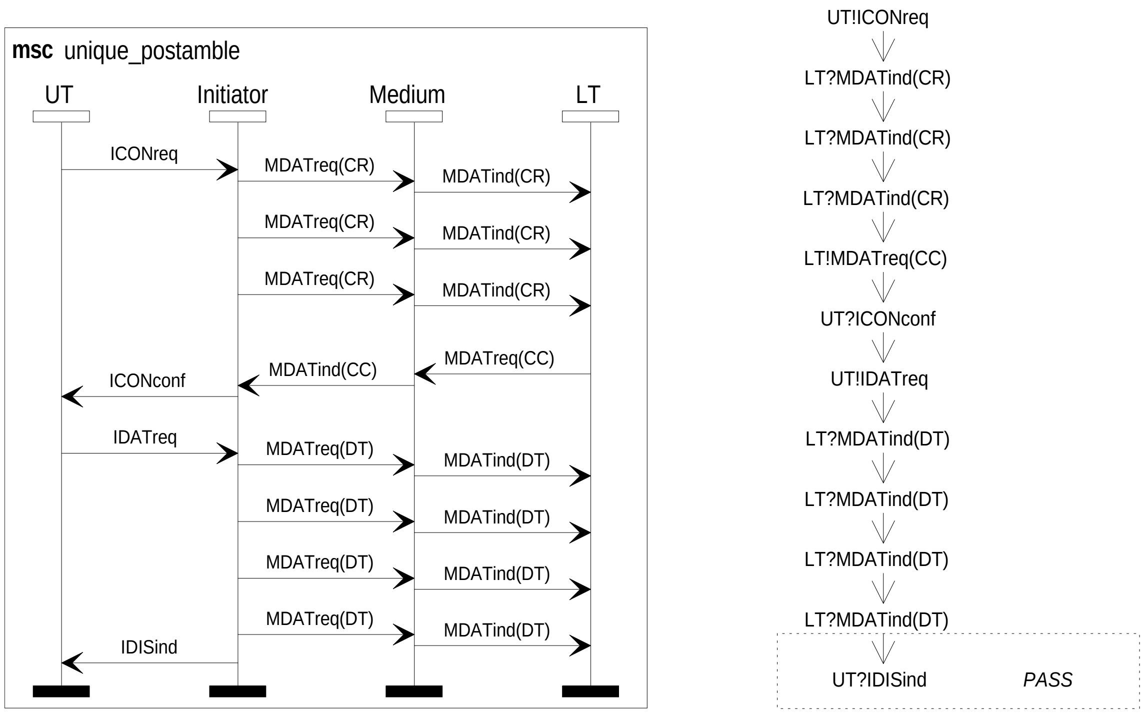
(b) An observable of (a)

Figure 6: Undesired system run

may exist several unique pass observables for one test purpose. For the test case we choose a subset of the shortest unique pass observables as pass observables.[15]

For our example Figure 7 (a) presents the trace which has a unique pass observable. Instead of a normal disconnection the upper tester UT initiates a data transfer by sending a data request IDATreq. After receiving the IDATreq the Initiator transmits

[15]We like to state clearly that we define three sorts of pass observables. The possible pass observables which are described in step 1, the unique pass observables which are defined in this paragraph and pass observables of the test case. The pass observables of a test case are the unique pass observables which can be found in the test case description.



(a) MSC (b) Unique pass observable

Figure 7: MSC of Figure 4 with unique postamble

a DT to the lower tester LT and waits for an acknowledgement. The lower tester LT does not answer and, therefore the DT is retransmitted three times. Afterwards, the Initiator indicates the failed data transmission by sending a IDISind to the upper tester UT and returns to its initial state.

We choose the observable of the described trace as the pass observable of the test case which should be generated (cf. Figure 7 (b)). From this the TTCN test case shown in Figure 8 follows. The pass observable can be found in the lines 1 to 12.

3. In a third step the *inconclusive observables* are calculated. An inconclusive observable has a common prefix with a pass observable, but it ends with an allowed response of the IUT from which one can conclude that the required pass observable is not performed. The inconclusive observables of our example can be found within the TTCN description in Figure 8.

4. In the fourth and last step the *fail observables* are defined. They need not to be calculated, because it is assumed that the allowed behavior is described by an SDL specification. Hence, every deviation from the SDL specification is wrong. In TTCN this can be easily defined by means of a default behavior description. The default of our example is shown in Figure 9.

In the previous two sections the whole SaMsTaG method has been introduced. An overall view of the SaMsTaG method is shown in Figure 10. Further information can be found in [12], [13], [32] and [31].

Test Case Dynamic Behaviour					
Test Case Name	**:** Test_Case_Example				
Group	**:**				
Purpose	**:** Connection establishment after the third retransmission of a Connection Request				
Default	**:** Unexpected_Events				
Comments	**:**				
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		UT!ICONreq			
2		LT?MDATind(CR)			
3		LT?MDATind(CR)			
4		LT?MDATind(CR)			
5		LT!MDATind(CC)			
6		UT?ICONconf			
7		UT!IDATreq			
8		LT?MDATind(DT)			
9		LT?MDATind(DT)			
10		LT?MDATind(DT)			
11		LT?MDATind(DT)			
12		UT?IDISind		PASS	
13		LT?MDATind(CR)		INCON	
14		LT?MDATind(CR)		INCON	
Detailed Comments :					

Figure 8: A TTCN test case

Default Dynamic Behaviour					
Default Name	**:** Unexpected_Events				
Group	**:**				
Objective	**:** Handle unexpected events				
Comments	**:**				
Nr	**Label**	**Behaviour Description**	**Constraints Ref**	**Verdict**	**Comments**
1		UT?OTHERWISE		FAIL	
2		LT?OTHERWISE		FAIL	
Detailed Comments:					

Figure 9: TTCN default behavior for the test case in Figure 8

6 The SAMSTAG tool

The SAMSTAG tool realizes the SAMSTAG method for specifications written in SDL and test purposes defined by MSCs. The tool architecture is shown in Figure 11. The SAMSTAG tool consists of an *MSC simulation tool*, an *SDL simulation tool* and a *test case generator*. The front- and backends are commercial SDL, MSC and TTCN editors.

The MSC simulation tool consists of an *MSC transformer* and an *MSC interpreter*. The MSC transformer transforms the MSC input into an internal data structure which, during test case generation, is interpreted by the MSC interpreter.

For reasons of performance, the SDL simulation tool is implemented in a different way. It consists of an *SDL transformer* which transforms an SDL specification into an executable C++ program, the *SDL simulator*. The SDL simulator behaves like the spec-

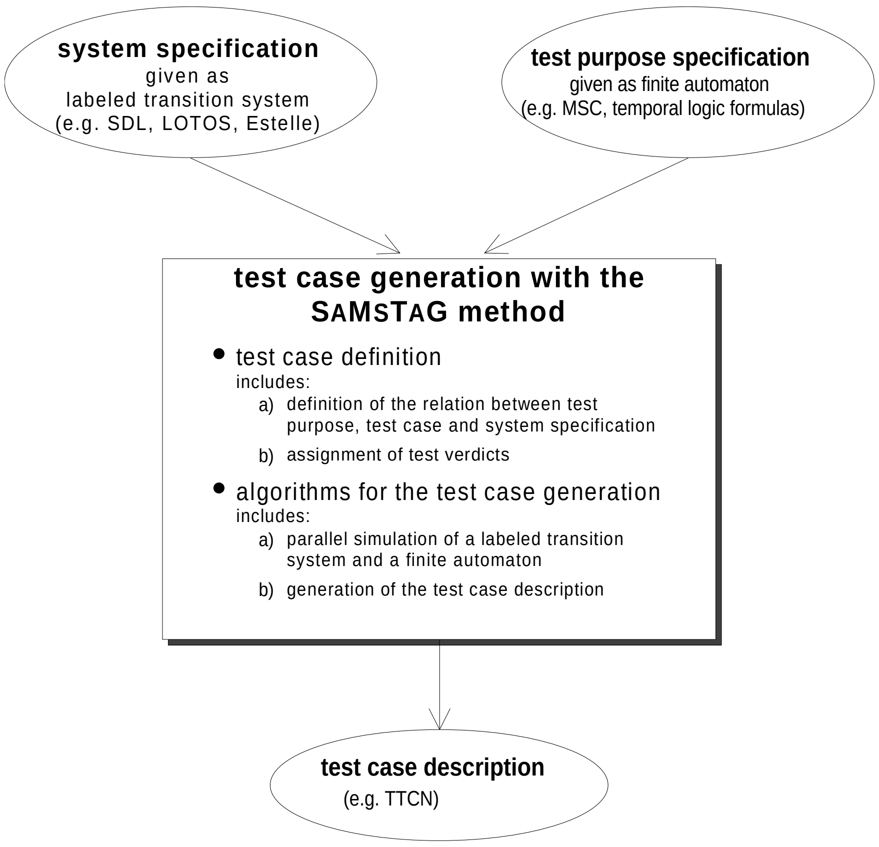

system specification
given as
labeled transition system
(e.g. SDL, LOTOS, Estelle)
test purpose specification
given as finite automaton
(e.g. MSC, temporal logic formulas)
test case generation with the SAMSTAG method
test case definition
includes:
a) definition of the relation between test purpose, test case and system specification
b) assignment of test verdicts
algorithms for the test case generation
includes:
a) parallel simulation of a labeled transition system and a finite automaton
b) generation of the test case description
test case description
(e.g. TTCN)

Figure 10: The SAMSTAG method

ification.

The test case generator controls the MSC interpreter and the SDL simulator. During test case generation it calculates the possible pass observables, the unique pass observables and the inconclusive observables. Finally, the test case generator defines the fail observables and stores the TTCN description of the generated test case as ASCII file.

6.1 The calculation of the possible pass observables

The computation of the possible pass observables is a typical search problem. The test case generator has to find SDL traces which include the events specified by the MSC and which lead the SDL system from its initial state back to its initial state. The observable of such a trace is a possible pass observable. Unfortunately, we can not ensure that we find possible pass observables, because this problem is equivalent to the halting problem of Turing machines [2] which is not decidable [20]. One only can search and hope to find the required observables. We search by simulating the SDL description and the MSC in parallel.

There exist several search methods like depth and breadth search. Breadth search can not applied because it is impossible to store all reached states of the SDL system[16]. Also

[16]A state of an SDL system includes the control states of the processes, the contents of the queues and the values of the variables.

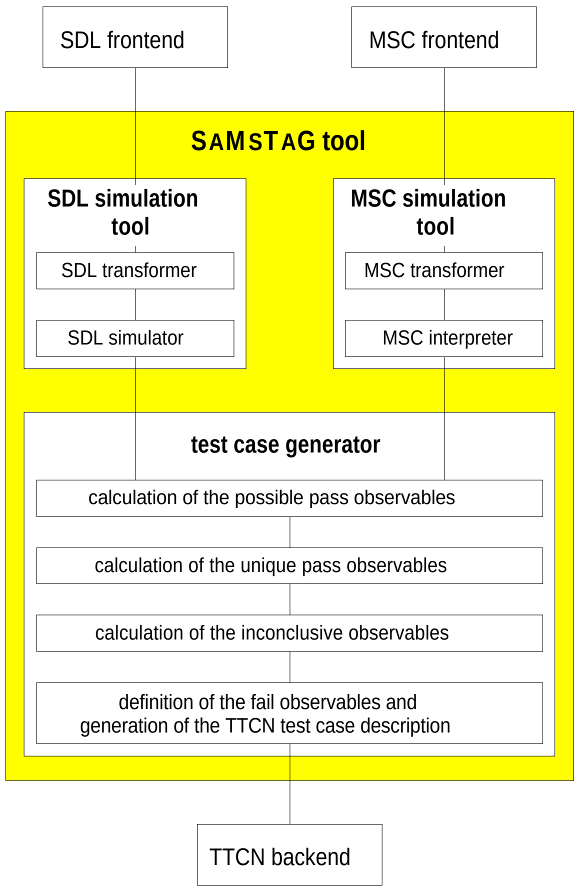

SDL frontend
MSC frontend
SAMSTAG tool
SDL simulation tool
SDL transformer
SDL simulator
MSC simulation tool
MSC transformer
MSC interpreter
test case generator
calculation of the possible pass observables
calculation of the unique pass observables
calculation of the inconclusive observables
definition of the fail observables and generation of the TTCN test case description
TTCN backend

Figure 11: The architecture of the SAMSTAG tool

depth search is not usable since we can not guarantee termination. As a consequence we use a *k-bounded depth search* which evaluates all possible traces of length k. If no trace with the required properties is found, the search can be repeated with a higher bound k or stopped without results.

6.2 The calculation of the unique pass observables

Also the unique pass observables are calculated by simulation. For each possible pass observable the traces which have the possible pass observable as observable are generated. Then, it is checked whether all of them fulfill the conditions (a) and (b) on page 13. In general, there may exist many unique pass observables. In this case we select a subset of the shortest unique pass observables to be the pass observables of the generated test case. Similar to the possible pass observables the existence of unique pass observables can not be guaranteed. There may exist possible, but no unique pass observables.

6.3 The calculation of the inconclusive observables

For the chosen unique pass observables the corresponding inconclusive observables have to be generated. Therefore the SDL description is simulated according to the traces of the pass observables. An inconclusive observable ends in a response of the IUT from which one can conclude that the required pass observable is not performed.

6.4 Fail observables and generation of the TTCN description

Finally, the pass and inconclusive observables are transformed into the TTCN notation and the fail cases are added by means of a TTCN default behavior description. The result of the test case generation is an ASCII file which includes the TTCN description of the generated test case.

7 Summary and outlook

In the previous sections the SAMsTAG method and the SAMsTAG method have been introduced. The SAMsTAG method makes it possible to generate abstract test cases for conformance tests based on a formal specification and a set of test purposes. For this the SAMsTAG method formalizes the term *test purpose* which is an important concept in CTMF/FMCT. Furthermore, in Section 2 the CTMF/FMCT conformance testing procedure is discussed and compared with other methods for test case generation.

All described approaches formalize different steps of the entire conformance testing procedure. Figure 12 presents an overall view of the parts which are formalized by the different approaches. The approaches may intersect in several aspects, but for the sake of clearness we omitted all overlaps.

Figure 12 also shows, that the theoretical foundation for the *test purpose definition* (1) has not been worked out. But, we believe that it is possible to formalize this step by means of the coverage criteria which already are used in the scientific methods.

However, for the application of a complete formal model for conformance testing to practice further investigations might be necessary. Particularly, the concepts *coverage criterion* (j) and *conformance relation* (h) have to be generalized and adopted to practical needs.

Acknowledgements

The elaboration of this report was funded by the KWF-Project No. 2555.1 *'Graphical Methods in the Test Process'*. The author would like to thank Beat Brändle, Dieter Hogrefe, and Roger Schönberger for supporting this work, Jan Kroon and Ekkart Rudolph for proofreading, and Matthias Günter, Peter Gurtner, Robert Nahm, Kurt Neuenschwander, Andreas Spichiger, and Daniel Toggweiler for valuable comments and suggestions.

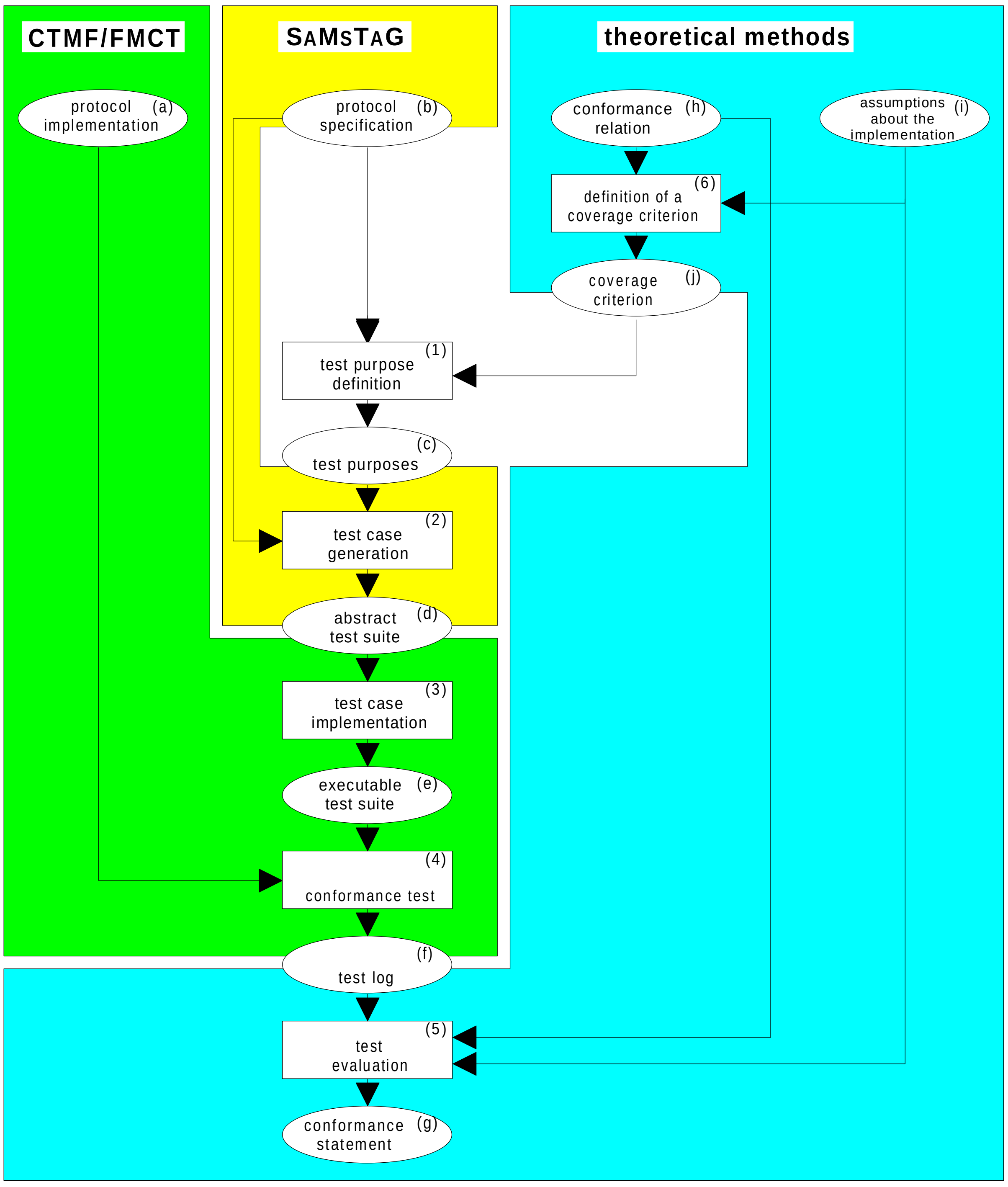

CTMF/FMCT
SAMSTAG
theoretical methods
protocol implementation (a)
protocol specification (b)
conformance relation (h)
assumptions about the implementation (i)
definition of a coverage criterion (6)
coverage criterion (j)
test purpose definition (1)
test purposes (c)
test case generation (2)
abstract test suite (d)
test case implementation (3)
executable test suite (e)
conformance test (4)
test log (f)
test evaluation (5)
conformance statement (g)

Figure 12: A complete formal description of the conformance testing procedure

References

[1] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. The BCS Practitioner Series, Series editor: R. Welland. Carl Hanser Verlag and Prentice Hall International, 1991.

[2] D. Brand and P. Zafiropulo. On communicating finite state machines. *Journal of the Association for Computing Machinery*, 30(2), April 1983.

[3] E. Brinksma. A Theory for the Derivation of Tests. In *Proceedings of the IFIP WG 6.1 Eighth International Symposium on Protocol Specification, Testing and Verification*. Elsevier Science Publishers B.V., June 1988.

[4] J. Grabowski, P. Graubmann, and E. Rudolph. The Standardization of Message Sequence Charts. In *Proceedings of the IEEE Software Engineering Standards Symposium 1993*, September 1993.

[5] J. Grabowski, M. Günter, P. Gurtner, D. Hogrefe, R. Nahm, K. Neuenschwander, A. Spichiger, and D. Toggweiler. Conformance Testing - A Tool for the Generation of Test Cases. Second Interim Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, June 1993.

[6] J. Grabowski, M. Günter, P. Gurtner, D. Hogrefe, R. Nahm, K. Neuenschwander, A. Spichiger, and D. Toggweiler. Conformance Testing - Ein Werkzeug zur Generierung von Testfällen. Eine Zusammenfassung der Projektergebnisse und ein Projektausblick. Project Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, September 1993.

[7] J. Grabowski, M. Günter, P. Gurtner, D. Hogrefe, R. Nahm, K. Neuenschwander, A. Spichiger, and D. Toggweiler. Conformance Testing - Ein Werkzeug zur Generierung von Testfällen. Abschlussbericht Teil 1: Eine Zusammenfassung der Projektergebnisse. Final Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, November 1993.

[8] J. Grabowski, M. Günter, P. Gurtner, D. Hogrefe, R. Nahm, K. Neuenschwander, A. Spichiger, and D. Toggweiler. Conformance Testing - Ein Werkzeug zur Generierung von Testfällen. Abschlussbericht Teil 2: Die SAMSTAG Methode. Final Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, November 1993.

[9] J. Grabowski, M. Günter, P. Gurtner, D. Hogrefe, R. Nahm, K. Neuenschwander, A. Spichiger, and D. Toggweiler. Conformance Testing - Ein Werkzeug zur Generierung von Testfällen. Abschlussbericht Teil 3: Das SAMSTAG Werkzeug. Final Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, November 1993.

[10] J. Grabowski, M. Günter, P. Gurtner, D. Hogrefe, R. Nahm, K. Neuenschwander, A. Spichiger, and D. Toggweiler. Conformance Testing - Ein Werkzeug zur Generierung von Testfällen. Abschlussbericht Teil 4: Eine Fallstudie zu Q.921. Final Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, November 1993.

[11] J. Grabowski, D. Hogrefe, P. Ladkin, S. Leue, and R. Nahm. Conformance Testing - A Tool for the Generation of Test Cases. First Interim Report of the F & E Project, Contract No. 233/257, Funded by Swiss PTT, May 1992.

[12] J. Grabowski, D. Hogrefe, and R. Nahm. A Method for the Generation of Test Cases Based on SDL and MSCs. Technical Report IAM-93-010, Universität Bern, Institut für Informatik, April 1993.

[13] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In O. Faergemand and A. Sarma, editors, *SDL'93 - Using Objects*. North-Holland, October 1993.

[14] J. Grabowski, D. Hogrefe, R. Nahm, and A. Spichiger. Relating Test Purposes to Formal Specifications: Towards a Theoretical Foundation of Practical Testing. Technical Report IAM-93-014, Universität Bern, Institut für Informatik, June 1993.

[15] J. Grabowski, R. Nahm, A. Spichiger, and D. Hogrefe. Die SAMSTAG Methode und ihre Rolle im OSI Konformitätstesten. Technical Report IAM-93-024, Universität Bern, Institut für Informatik, October 1993.

[16] D. Hogrefe. *Estelle, LOTOS und SDL - Standard Spezifikationssprachen für verteilte Systeme*. Springer Verlag, 1989.

[17] D. Hogrefe. OSI Formal Specification Case Study: The Inres Protocol and Service (revised). Technical Report IAM-91-012, Universität Bern, Institut für Informatik, May 1991, Update May 1992.

[18] D. Hogrefe. On the Development of a Standard for Conformance Testing based on Formal Specifications. *Computer Standards & Interfaces 14*, 1992.

[19] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, Inc., 1991.

[20] J. E. Hopcroft and J. D. Ullmann. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[21] ISO, E. Brinksma (ed.). Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observable Behaviour. International Standard 8807, ISO, Geneva, 1988.

[22] ISO/IEC JTC 1/SC 21 N. Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework. International Multipart Standard 9646, ISO/IEC, 1992.

[23] ISO/IEC JTC 1/SC 21/WG 1. Open Systems Interconnection - Data Management and Open Distributed Processing - Working Draft on Fomal Methods in Conformance Testing. Technical report, ISO/IEC, July 1993.

[24] ISO/IEC JTC 1/SC21. Information Technology - Open Systems Interconnection - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation. International Standard 9646-3, ISO/IEC, 1992.

[25] ITU Telecommunication Standards Sector SG 10. ITU-T Recommendation Z.100: Functional Specification and Description Language (SDL) (formerly CCITT Recommendation Z.100). ITU, Geneva, June 1992.

[26] ITU Telecommunication Standards Sector SG 10. ITU-T Recommendation Z.120: Message Sequence Chart (MSC). ITU, Geneva, June 1992.

[27] L. Kahn. State of Research in the Area of Formal Test Specification Methods. Draft Technical Report ATM - 1006 - 1, ETSI TC ATM, 1991.

[28] J. Kroon and A. Wiles. A Tutorial on TTCN. In *Proceedings of the 11th International IFIP WG 6.1 Symposium on Protocol, Specification, Testing and Verification*, 1991.

[29] Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, 1990. 1990 ACM-0-89791-404-X/90/0008/3777.

[30] R. Nahm. Semantics of Simple SDL. In H. König, editor, *GI/ITG Fachgespräch: 'Formale Beschreibungstechniken für verteilte Systeme' in Magdeburg (Germany)*, volume 8 of *FOKUS-Band*. K.G. Saur-Verlag, 1993.

[31] R. Nahm. *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts*. PhD thesis, University of Berne, Institute for Informatics and Applied Mathematics, February 1994.

[32] R. Nahm, J. Grabowski, and D. Hogrefe. Test Case Generation for Temporal Properties. Technical Report IAM-93-013, Universität Bern, Institut für Informatik, June 1993.

[33] Swedish Telecom, S-123 86 Farsta. *ITEX-DE version 2.0*, 1992.

[34] P. Wolper. On the Relations of Programs and Computations to Models of Temporal Logic. In *Proceedings Temporal Logic in Specification*, Lecture Notes in Computer Science 398, 1989.